ADA275260

Program Derivation by Proof Transformation

Penny Anderson

October 1993

CMU-CS-93-206

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis Committee:

Frank Pfenning, Chair
David Garlan
Dana Scott
Robert Constable, Cornell University

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597. The author holds a Bourse Chateaubriand from the Government of France.

Carnegie Mellon

School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

Program Derivation by Proof Transformation

PENNY ANDERSON

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

Frank Pfenning — THESIS COMMITTEE CHAIR — October 29, 1993 — DATE

[illegible] — DEPARTMENT HEAD — 10/29/93 — DATE

APPROVED:

[illegible] — DEAN — 10/30/93 — DATE

Abstract

In the proofs-as-programs methodology, verified programs are developed through theorem-proving in a constructive logic. Under this approach, the theorem-proving process can be regarded as a program derivation process. The merits of this approach to programming are twofold. First, working with proofs instead of programs concentrates the developer's effort on the intellectually difficult part of the development process: understanding, solving, and explaining the solution to a mathematical problem. Second, the proof provides a formal and trustworthy basis for an explanation of the program. This thesis investigates the use of proof transformations as a way to address important concerns in program derivation that are not addressed by theorem-proving alone.

One difficulty with the proofs-as-programs strategy arises from the conflict between elegance and efficiency. A simple, elegant proof may lead to an inefficient program. A more complex proof that corresponds to a more efficient program may be difficult to invent or understand. With proof transformations, a developer can start with an elegant proof that is easy to understand, and incrementally derive a more complex proof and thus a more efficient program. Another problem comes from the need for adaptation and reuse. With current automated support for theorem-proving, it is difficult to re-use previous work other than by re-using lemmas from a library. This kind of reuse is analogous to the use of subroutine libraries in ordinary programming, and does not directly support adaptation. Proof transformations provide a way of adapting a proof to a new context.

One standard approach to metaprogramming tasks like proof transformation has been to use a separate programming language, such as ML, as a metalanguage for a type theory considered as an object logic. A more recently developed strategy, which has been applied to programming language semantics, theorem-proving, and related problems, is the use of a higher-order logic programming language as a logical framework. This thesis adopts the second approach, using the Elf programming language, which gives a logic programming interpretation to the Edinburgh Logical Framework. We show a partially verified implementation of support for the proofs-as-programs strategy and proof transformations, and argue that the implementation techniques contribute to the concise, declarative, and verifiable implementation of metaprogramming tasks for formal logic. Through case studies of small programming problems, we demonstrate that known program transformations can be implemented in the domain of proofs, and expressed as derived logical rules. The case studies supply evidence that a development methodology based on proof transformation can provide a useful integration between the flexibility of program transformation and the formal connection between a program and its specification of the proofs-as-programs methodology.

Thesis Committee: **Frank Pfenning,** Thesis Advisor
Carnegie Mellon University

Robert Constable
Cornell University

David Garlan
Carnegie Mellon University

Dana Scott
Carnegie Mellon University

Author's present address: Penny Anderson
Projet CROAP
INRIA
2004, Route des Lucioles
06902 Sophia Antipolis CEDEX
France

To all my parents

Contents

List of Figures

Chapter 1

Introduction

Research in constructive logics and type theories has resulted in a good understanding of the relation of formal constructive proof to computation, and has produced a number of systems (e.g., Nuprl [C+86], the Calculus of Constructions [DFH+91], Alf [Mag92], LEGO [LPT89], PX [Hay90]) capable of supporting the development of programs by constructive theorem-proving (the *proofs-as-programs* strategy). There are now a number of researchers experimenting with the methodology, but there are many problems that have to be solved before it will achieve applicability to practical software development.

This thesis continues a line of research that began with the observation of Goad [Goa80] that a formal proof contains information that is not essential for computation, but that "is useful in the transformation of computing methods;" his work exploits this information to transform proofs in order to specialize algorithms to their inputs. Pfenning [Pfe90] generalizes this line of thought and argues that proof transformation can be used to address two problems with the pure proofs-as-programs methodology. The first problem is the trade-off between elegance and efficiency, which exists in the domain of proofs for much the same reasons that it exists in the domain of programs. The second is the need to adapt programs to changing specifications. A change to a specification imposes a need to change the associated proof; although theorem proving tools (proof editors, automated provers, etc.) can support this process to some extent, there are many difficulties that result in duplication of effort. In this thesis we investigate some ways to address these problems by proof transformation, using the Elf language [Pfe91a] to implement a small constructive logic and functional programming language, the extraction of programs from proofs, and transformations on proofs. We carry out several case studies of program development showing the application of these techniques.

In this introduction we describe the conceptual basis for the research. Section 1.1 gives a brief introduction to the proofs-as-programs strategy. Section 1.2 discusses the use of a type theory as a logical framework. Then we discuss proof transformations and give a small example adapted from Goad. Finally we make some general remarks about the aims, methods, and contributions of our work, discuss some related research, and describe the organization of the body of the thesis.

1.1 Proofs as programs

The development of verified programs through theorem-proving in a type theory or constructive logic has been explored by many researchers, *e.g.*, [C+86], [CH85], [Hay90], [MW81], [ML80], [C+86], [CH85], [Hay90]. Although various systems have been used to formalize this proofs-as-programs strategy, the basic idea is the same: to rely on a realizability interpretation of constructive logic in order to obtain a program from a proof. A verified program is obtained by the following process: write a specification for the program as a theorem in the language of a formal deductive system, prove the theorem in a constructive way, and finally extract a term in a functional programming language from the proof. Usually the specification has the form $\forall x : \tau \,.\, \exists y : \tau' \,.\, P(x, y)$; then the extracted term has the type $\tau \longrightarrow \tau'$, and it is a function that, given an x of type τ, computes a y of type τ' such that $P(x,y)$ holds.

A simple example, presented informally, is the following proof for an algorithm to compute an upper bound of both the sum and product of two rational numbers. The example is adapted from Goad [Goa80].

The specification is the theorem to be proved, which says that, given two rational numbers x and y, an upper bound for their sum and product exists.

Specification 1.1 $\forall x \,.\, \forall y \,.\, \exists z \,.\, (z \geq x + y) \ \wedge \ (z \geq xy)$

The proof of the theorem is a very simple case analysis.

Proof 1.2 There are two cases:

Case $x \leq 1$. Then let $z = y + 1$, since $(y + 1 \geq x + y) \ \wedge \ (y + 1 \geq xy)$.

Case $x > 1$. There are two subcases:

> Case $y \leq 1$. Then let $z = x + 1$, since $(x + 1 \geq x + y) \ \wedge \ (x + 1 \geq xy)$.
>
> Case $y > 1$. Let $z = 2xy$, since $(2xy \geq x + y) \ \wedge \ (2xy \geq xy)$.

□

Programs extracted from proofs closely follow the proof structure; here, the case analyses correspond to conditionals. For readability, we show programs in ML rather than λ-calculus. One possible realization for the proof is:

Program 1.3

```
fun u x y = if x <= 1 then y+1 else (if y <= 1 then x+1 else 2*x*y)
```

The virtues of this approach to program development have been described extensively elsewhere, notably by Bates and Constable [BC85]. Briefly, we can say that working with proofs instead of programs concentrates the developer's effort on the intellectually difficult part of the development process, i.e., understanding, solving, and explaining the solution to a mathematical problem. To the extent that the proof process and program extraction can be trusted, the program so developed is guaranteed to be a correct implementation of the formal specification. With an automated system to aid in the manipulation of proofs, the developer has significant help in carrying out this part of the development process, and the proof provides a formal basis for the documentation of the program.

In what sense does Program 1.3 give the computational content of Proof 1.2? Section 2.3 gives a precise description of our working notion of the computational content of a proof, but to set the stage, this section gives an intuitive explanation based on a constructive interpretation of logical operations, which originated with Heyting [Hey34]. Formal systems for defining realizability conform to this interpretation.

The interpretation relies on an informal notion of "construction". For an initial understanding, it is enough to take this to mean a function definable in a suitable programming language. (Constructions as terms in a typed lambda calculus – the *Curry-Howard isomorphism* – are described in [How69].) Then the following describes what it means to constructively prove a logically compound statement (this is adapted from [TvD88]):

- A proof of $A \wedge B$ is given by presenting a proof of A and a proof of B.
- A proof of $A \vee B$ is given by presenting either a proof of A or a proof of B.
- A proof of $A \supset B$ is a construction that when given any proof of A produces a proof of B.
- There is no proof of $\perp$ (absurdity, contradiction).
- A proof of $\neg A$ is a construction that when given any proof of A produces a proof of $\perp$.
- A proof of $\forall x \,.\, A(x)$ is a construction that when given a term t produces a proof of $A(t)$.
- A proof of $\exists x \,.\, A(x)$ is a witness t and a proof of $A(t)$.

This description gives a direct intuitive basis for the rules of Gentzen's (intuitionistic) natural deduction [Gen69], in the sense that it is evident how to compute a proof in the above sense for the conclusion of each rule, given constructive proofs for the premises. For example, the rule of implication introduction says that, given a proof of B under the assumption A, we can conclude $A \supset B$:

$$\dfrac{\begin{array}{c}\overline{A}^{\,p}\\ \vdots \\ B\end{array}}{A \supset B}\supset\mathrm{I}^{p}$$

But a given proof of B under the assumption A *is* a construction that transforms a proof of A into a proof of B, so it is also a proof of $A \supset B$ on the above interpretation. The rule of implication elimination says that, given a proof of $A \supset B$ and a proof of A, we can conclude B:

$$\dfrac{A \supset B \qquad A}{B}\supset\mathrm{E}$$

According to the interpretation, we have a construction that transforms any proof of A into a proof of B. Then a proof of B can be obtained by applying this construction to the given proof of A.

Now it should be possible to see how Program 1.3 represents the computational content of Proof 1.2. The end-formula of the proof is $\forall x \,.\, \forall y \,.\, \exists z \,.\, (z \geq x+y) \wedge (z \geq xy)$. The program defines a function of two arguments, which correspond to the two universally quantified variables of the formula. It returns a value that witnesses the existential. Strictly speaking, it should return a pair consisting of a value z and a proof p of $(z \geq x+y) \wedge (z \geq xy)$. Since p has no computational interest we have suppressed it. Section 2.3 describes how our implementation systematically suppresses some computationally uninteresting information using modified realizability.

The description also gives a basis for rejecting the classical principle of the excluded middle, i.e. $A \vee \neg A$ for any A. If the principle is accepted, the interpretation implies that we have either a proof of A or a proof of $\neg A$, i.e., every proposition is decidable. From an operational point of view, it is also useful to consider the equivalent law of indirect proof, or proof by contradiction:

$$\begin{array}{c} \overline{}^{\,p} \\ \neg A \\ \vdots \\ \dfrac{\bot}{A}\; LIP^{p} \end{array}$$

If this were acceptable, then in particular we could construct a proof of the form

$$\begin{array}{c} \overline{\neg \exists x \,.\, A(x)}^{\,p} \\ \vdots \\ \dfrac{\bot}{\exists x \,.\, A(x)}\; LIP^{p} \end{array}$$

But a constructive interpretation of the premise is a construction that transforms a proof of $\neg \exists x \,.\, A(x)$ to a proof of $\bot$. This does not give any way in general to compute a witness for the conclusion, that is, a t such that $A(t)$ is true.

So as not to mislead the reader, it is important to mention that there are ways of recovering computational content from certain classical proofs. Murthy's thesis [Mur90] describes how this can be done in practice and explores the very interesting connection to nonlocal control operators.

1.2 Type theory as a logical framework

Constructive approaches to mathematics go back at least to the work of Brouwer early in this century. The subtleties and implications of these approaches for pure mathematics have little relevance to this thesis; useful historical sketches can be found in [C^{+}86] and [TvD88]. The type theories currently being studied in computer science may be said to descend from the AUTOMATH family of languages [dB80] for the machine checking of mathematics, and from Martin-Löf's development of a formal theory for expressing the syntax and semantics of constructive mathematics [ML73].

In current computer science research there are two main ways of exploiting type theory. One is to use it as a formal representation of constructive logic to support, among other activities, programming by theorem proving. Since type theory provides an internal λ-calculus as well as a representation of logic, the computational contents of a proof can be internally represented. This is the approach of most work with Nuprl and the Calculus of Constructions, and there are type theories (e.g. PX) carefully tailored for use as programming languages. Another approach, that taken in this thesis, is to use a type theory as a *logical framework* or metalanguage for formally representing a logic. LF, the Edinburgh Logical Framework [HHP93] is a type theory designed for this purpose. Our implementation of proof transformations uses the programming language Elf [Pfe91a], which gives an operational interpretation to the types of LF to support a logic programming style of metaprogramming. The framework approach permits a freer choice of object logic and programming language than is possible when using a type theory directly as a logic and/or programming language, and leaves more control in the hands of the implementor (over reduction, for example). As a consequence, for an object logic we are able to choose a pure logic, since the framework provides a λ-calculus in which to represent the program extracted from a proof, and we give two different formulations of program extraction. However, this freedom and control have both an implementation cost and a theoretical cost. An implementation of a type theory will provide normalization of terms of that theory (which amounts to program execution) for free, and the metatheory which is worked out for the type theory gives theorems for free about the theory regarded as a logic or programming language. On the other hand, when the type theory is used as a framework, the logic and programming language being studied are not the language of the type theory. The implementor must write (meta) programs to execute programs of the language under consideration. The metatheory of the framework is not the metatheory of the logic, which must be developed explicitly.

1.3 Proof transformation

A *proof transformation*, for the purposes of this work, is a procedure that, given a formal proof $\mathcal{D}$, produces another formal proof $\mathcal{D}'$, where the validity of $\mathcal{D}$ (along with the correctness of the transformation) guarantees the validity of $\mathcal{D}'$, although $\mathcal{D}$ and $\mathcal{D}'$ may not prove the same theorem. The transformation tactics of Nuprl [C$^+$86] implement this idea. Because of the close relation between the structure of a proof and the program extracted from it, a transformation can be useful for the *form* of the proof it produces.

Proof transformations may depend on global analysis (as in [Mur90]), requiring inductive proofs of correctness. However, as we show here, useful changes to program structure may be accomplished by *uniform* proof transformations, i.e, those that depend only on local syntactic properties of the proofs. As will become apparent, in a natural deduction setting they can be thought of as derived rules of inference. When properly encoded in the Elf language they are proved correct by type checking. The simplest of these encoded transformations correspond to higher-order patterns in the sense of [Pfe91b]. Though often it is not possible to code a rule in this restricted form, the style of transformation we consider can still be encoded in the form of a single rewrite rule to obtain an executable Elf program, if the application of the rule is restricted so that only ground terms (complete closed proofs) are supplied as input. Thus the encodings are related to the work of Huet and Lang [HL78] on program transformations expressed as second-order patterns, translated to the proof level, and to that of Hannan and Miller [HM88].

1.3.1 An example

The following illustration of proof transformation applies Goad's pruning transformation [Goa80] to Proof 1.2. Pruning is a way of allowing for adaptation and reuse in the case of the specialization of a program to fixed values for its inputs. Here, proof transformation specializes the proof to $y = .5$ by substituting the value for the variable throughout the proof, normalizing the result, and then pruning.

Here is an informal presentation of the new specification and its proof after the substitution of .5 for y and normalization. The case distinction on the value of y has been eliminated.

Specification 1.4

$$\forall x\,.\,\exists z\,.\,(z \geq x + .5) \ \wedge \ (z \geq (x)(.5))$$

Proof 1.5 There are two cases:

Case $x \leq 1$. Then let $z = 1.5$, since $(1.5 \geq x + .5) \ \wedge \ (1.5 \geq (x)(.5))$.

Case $x > 1$. Now $.5 \leq 1$; therefore, let $z = x + 1$, since $(x + 1 \geq x + .5) \ \wedge \ (x + 1 \geq (x)(.5))$.

□

Program 1.6

(From Proof 1.5)

```
fun u' x = if x <= 1 then 1.5 else x+1
```

This program can be obtained from Program 1.3 by partial application and reductions in the λ-calculus. This seems to be the best one can hope for by operating purely on the program without looking at the proof. But it is evident from the proof that the case analysis on x is unnecessary: when $y = .5$, $x + 1$ satisfies the specification regardless of the value of x. Goad's pruning transformation provides a way to capture this observation because it transforms the proof rather than the program, allowing us to change the function computed, not just specialize it to one of its inputs, while ensuring that the program satisfies the specification.

The particular form of pruning needed for this example replaces a case analysis by one of its arms when the case that holds for that arm is not used. We show the transformation as a schema for proofs in natural deduction style.

Transformation 1.7 *Transform*

$$\dfrac{\begin{array}[b]{c}\mathcal{D}_1\\ \vdots\\ A \vee B\end{array} \qquad \begin{array}[b]{c}\overline{A}^{\,p}\\ \vdots\\ \mathcal{D}_2\\ \vdots\\ C\end{array} \quad \begin{array}[b]{c}\overline{B}^{\,p}\\ \vdots\\ \mathcal{D}_3\\ \vdots\\ C\end{array}}{C}\,\vee\mathrm{E}^{p}$$

to

$$\begin{array}{c}\mathcal{D}_3\\ \vdots \\ C\end{array}$$

provided that B does not appear as an open assumption in $\mathcal{D}_3$.

This transformation is applicable to Proof 1.5 because the assumption $x > 1$ is not needed in order to show that $(x + 1 \geq x + .5) \wedge (x + 1 \geq (x)(.5))$. The result of the transformation is the trivial proof:

Proof 1.8 (of Specification 1.4) We pick $z = x + 1$, since $(x + 1 \geq x + .5) \wedge (x + 1 \geq (x)(.5))$ for all x. □

This corresponds to the program

Program 1.9

```
fun u'' x = x+1
```

Note that Program 1.9 and Program 1.6 compute different functions, although both satisfy Specification 1.4.

1.3.2 Proof transformations and program transformations

As the example shows, proof transformation shares many features with program transformation. Some proof transformations merely translate program transformations into a different domain. For instance, the substitution and normalization transformations that produce Proof 1.5 from Proof 1.2 correspond to simple program transformations. More significantly, both approaches emphasize a development process that proceeds by small, intellectually manageable increments, and both aim at supporting re-use of the objects being developed. But the pruning operation demonstrates a major difference between the two methodologies. A program describes the *functionality* of a computation, but not its *purpose*; by contrast, a proof describes both: the functionality is specified by the structure of the proof, and the purpose is expressed by the theorem that is proved. This added expressiveness allows the pruning transformation to produce Program 1.9, with its different functionality, from (the proof of) Program 1.6.

In general, proof transformations can, like pruning, both exploit and preserve more information than pure program transformations can. By "pure" program transformations we mean those that exploit only the information inherent in the text of the program. They may alter functionality, but only in very constrained ways like specialization. Since a theorem (*i.e.*, the specification) may have many different proofs, with corresponding programs of different functionality, proof transformation has the potential to produce a final implementation that computes a different function from that of the initial implementation, as the pruning example shows. A proof transformation may even alter the specification: its validity consists in the property that, if the original proof $\mathcal{D}$ is a valid proof of some proposition P, then the resulting proof $\mathcal{D}'$ is a valid proof of some proposition P', where P and P' need not be the same. Translated into the programming world, this means that a proof transformation can give us a program for the specification P', given a program for the specification P.

1.4 Objectives of the thesis

The general motivation of this work is the hope of bringing the proofs-as-programs strategy closer to application to real programming tasks. We set out to explore how proof transformation may be used to attack some of the obstacles to this goal, in particular those noted by Pfenning: the tradeoff between elegant proofs and efficient programs, and the need to adapt to changing specifications. Along the way we have learned much about the strengths and weaknesses of a logical framework-based approach to formulating proof transformations.

The principles that have guided this research are that first, it is best to separate proof transformation from the problem of theorem proving; and second, proof transformations should be expressed in an understandable and verifiable way. These principles are motivated by the following observations.

Research on automated support for theorem proving is an active area, and there are several competing approaches, each with its strengths and weaknesses. As a research strategy, the separation of proof transformation and theorem proving is valuable as a way to isolate the contributions transformation may offer to problem solving. We should note however that our approach is predicated on the existence of *proof objects* of a particular kind. These are not available from all provers (e.g. the Boyer-Moore Theorem Prover [BM79]) although in principle there seems to be no reason why they could not be constructed.

The idea that transformations should be expressed in an understandable way is, we hope, non-controversial. Understandability is a widely accepted goal for programs in general, thus for the metaprograms that implement proof transformations. But understandability of proof transformations may also have importance for the documentation of the extracted program. In a pure proofs-as-programs strategy without transformation, the proof is considered to be the formal basis for an explanation of the program. But the tradeoff between elegance and efficiency intervenes: because of its complexity, the proof of a highly optimized program may not be much good as an explanation. An automatically synthesized (by transformation and theorem proving) proof is also not an explanation if the proof is too complex and the synthesis process is not transparent. Thus we want the proof transformations as part of the explanation of the program. Our work exploits the logical framework approach, and declarative aspects of the logic programming style, to achieve executable proof transformations that can be read as derived logical rules.

The verifiability of transformations is perhaps less critical, since the program resulting from a transformation is verified by a combination of proof checking and program extraction. But in principle transformations should be correct in general; moreover, it is often useful to know something more than that a transformation produces a valid proof. We may want to know also the end-formula of the proof, or some structural feature of the proof. The use of a logical framework contributes to these goals. We give transformations that can be encoded as derived rules, and thus verified *in general* by a proof checker (the type checker of the framework).

1.5 Research contributions

The thesis makes contributions in two main areas: the formalization of proofs-as-programs and proof transformations as a set of deductive systems in a logical framework, and the formulation of program transformations as proof transformations.

We give a partially verified implementation in Elf of basic support for the proofs-as-programs methodology: a constructive logic, a functional programming language, and the extraction of programs from proofs. The ideas behind the implementation and verification are not new (see Section 1.6 on related work for sources); our contribution has been to synthesize them in a working implementation and to use it to develop and transform proofs and extract programs. Our implementation of proof transformations is limited to those that can be expressed as derived rules of the object logic, although there is nothing in the underlying implementation that prevents more complex analysis and transformation. This formulation is declarative, easily verifiable, readily extensible, and independent of the theorem-proving problem. The implementation as a whole is partial evidence for the feasibility of the metalogical framework approach to metaprogramming for logic advocated by Basin and Constable in [BC93], although our approach is a hybrid, combining the weak framework LF and the logic programming capabilities of Elf.

We have shown that, perhaps surprisingly, useful program transformations translated to the level of proof transformation can be formulated as derived rules of the object logic. We have done this for two well-known program transformations. This suggests that the extensive research effort that has been devoted to codifying program transformations can be rather easily translated into the domain of proof transformation. An alternative view is that proof transformation techniques can be used to enrich program transformation by providing a simple syntactic formal representation of the semantic information that justifies a transformation. Our case studies also provide more evidence for Goad's thesis that proof transformation can achieve results not available from purely syntactic program transformation.

1.6 Related work

In his thesis Peter Madden [Mad91] also extends the work of Goad along the lines suggested by Pfenning's paper. He describes a complete Prolog-based reimplementation of Goad's pruning transformation, extends it to an implementation by proof transformation of the tupling program transformation strategy, and sketches an extension to divide-and-conquer transformations. This work aims at full automation, including automatic theorem proving as well as selection of proof transformation strategies. Thus there is a major difference in philosophy: where we are concerned to separate transformation from proving and from heuristic concerns, and to express transformations as transparently as possible, Madden's work emphasizes automated deduction.

Chetan Murthy's thesis [Mur90] studies translations from classical to constructive logic, which allow programs to be extracted that capture the computational content of certain classical proofs. His work focuses on the relationship between classical reasoning and nonlocal control operations, and the exploitation of the relationship to obtain total-correctness proofs of programs using nonlocal control. Unlike the simple uniform transformations we study here, Murthy's translation is defined by structural induction on proof trees, and considerable engineering expertise was required to obtain a feasible implementation. This implementation takes the standard approach of metaprogramming in ML, rather than the framework-based approach of our work.

Christophe Raffalli, in [Raf93], describes the use of proof translation to compile functional programs for an SEC abstract machine. He gives a type system and an operational semantics for the abstract machine, shows the soundness of the semantics with respect to the type system, and gives a translation from natural deduction-style proofs in second-order logic to the machine logic.

Working in an essentially "linear" context, Sieg and Wainer [SW93] have investigated proof translations in which recursive-to-tail-recursive program transformation corresponds to the elimination of what they call "call-by-value" cuts.

The basic techniques used in our Elf implementation were developed in [HM89], [Han91], [HHP93], and [MP91]. The partial internalization of the metatheory uses methods described in [HP92], [PR92], and [Pfe92a].

The procedure we use for optimizing code during program extraction is directly derived from techniques developed by Hayashi [Hay90] for PX, Paulin-Mohring [PM89] for the Calculus of Constructions, and Sasaki [Sas86] for Nuprl. Goad [Goa80] and Schwichtenberg [Sch82], [Sch85] use similar ideas to reduce the size of the representation of the computational contents of a proof.

1.7 Outline of the thesis

There are two main parts to the thesis: a description of the Elf implementation of proofs-as-programs and proof transformations, and some case studies of graph search algorithms.

- Chapter 2 gives a short introduction to the Edinburgh Logical Framework and the Elf language, then describes the Elf encoding of proofs, programs, and the extraction of programs.
- Chapter 3 presents some of the metatheory of extraction, with a partial formalization of it in Elf.
- Chapter 4 demonstrates by example a method of encoding and applying a limited class of proof transformations. A transformation for converting a program to tail-recursive form is given, first informally, then fully formalized in Elf. Then it is applied to a small programming problem.
- Chapter 5 presents some case studies, applying proof transformation to breadth-first and depth-first search. Here the emphasis is on the program development problem rather than on the formulation of the proof transformations.
- Chapter 6 gives a summary and a discussion of directions for future research.
- In the appendices we provide listings of Elf code with a description of how to access our implementation and the Elf system by ftp, and some technical remarks on the adequacy of Elf encodings and the recognition of tail-recursive object programs.

1.8 Acknowledgments

I am greatly indebted to my thesis advisor Frank Pfenning for his technical advice, encouragement, and matchless patience. He has given generously of his time, and without his ideas and support this work would not exist.

I would like to thank my thesis committee members: Robert Constable, David Garlan, and Dana Scott for their time, support, and expertise.

I thank my former advisor Bill Scherlis, who introduced me to the ideas that motivated me to do this work, for his patience and enthusiasm.

Gilles Kahn has given generously of moral, intellectual, and practical support during my stay at INRIA Sophia-Antipolis. His enthusiasm and perspective have been invaluable.

The faculty and students affiliated with the Ergo project and its offshoots have done much to educate and stimulate me during my time at CMU. The members of Gilles Kahn's research group at INRIA have likewise contributed to my intellectual growth in the last year. Carnegie Mellon's School of Computer Science provided a unique educational framework and very generous support over the years.

Sharon Burks's advice, practical aid, and good cheer proved indispensable for success in the CMU CS graduate program. Everyone who has been in my shoes knows how much I mean by this; others will simply have to imagine.

I particularly thank the following people who provided thoughtful discussions and/or moral support: Yves Bertot, Anne Candreva, Lori Cole, Anne-Marie Dery, Joelle Despeyroux, Thierry Despeyroux, Scott Dietzen, Susan Finger, Bob Harper, Francesmary Modugno, Scott Nettles, Benjamin Pierce, Phyllis Reuther, John Reynolds, Ellen Siegel, Kathryn Swedlow, and Angelika Zobel.

Finally I thank all the members of my family.

Chapter 2

Implementation

This chapter describes how we implemented some basic support for proofs-as-programs in a logical framework, and describes techniques of higher-order logic programming for defining inference systems over languages represented using higher-order abstract syntax. We encoded a natural-deduction style constructive logic and a programming language in the Edinburgh Logical Framework (LF) [HHP93] using Elf [Pfe89], [Pfe91a], a logic programming language based on LF. Program extraction, program execution, proof transformations, and other forms of term manipulation were implemented as Elf logic programs.

Section 2.1 gives a brief introduction to the LF and Elf languages and the principles guiding their use for encoding inference systems. Section 2.2 describes the encoding of natural deduction proofs for a first-order constructive logic. Finally in Section 2.3 we treat program extraction, defining a small functional programming language with a type assignment system and interpreter, and two forms of program extraction from proofs.

2.1 LF and Elf

LF is a typed λ-calculus designed to serve as a framework for the encoding of logics and related formal systems. Its type system is expressive enough to support straightforward encodings of many (though not all) formal deductive systems used in reasoning about programming languages, formal logics, and the like. The decidability of the type system gives practical force to the *judgments as types* and the corresponding *proof-checking as type-checking* principles. Elf is a logic programming language that gives an operational semantics to LF type declarations: an Elf program is a collection of type declarations and an Elf query is a type. The query may contain free variables, which are treated like Prolog logic variables; it succeeds if the system can construct an inhabitant of the query type from the declarations that constitute the program.

Elf signatures (collections of type declarations) may be used as language definitions, as logic programs, or both. In a language definition, types and type declarations are used in a way that corresponds to the usual intuition: they are syntactic categories and declarations of constructors for those categories. Signatures used in this way are called *static* and are not used for search. *Dynamic* signatures are programs used for search, which implement deductive systems for making certain kinds of judgments. This dual nature of Elf signatures reflects the LF principle of judgments-as-types.

2.1.1 The LF type system

First, we briefly describe the LF type theory. Harper et al. [HHP93] give a full description; here we give the syntax and sketch a few crucial ideas but rely on examples to develop a feeling for the semantics. LF is a three-level dependently-typed λ-calculus of *objects*, *families*, and *kinds*. Families classify objects and include ordinary types; kinds classify families.

$$\begin{array}{llll} \textit{Kinds} & K & ::= & \text{Type} \mid \Pi x{:}A\,.\,K \\ \textit{Families} & A & ::= & a \mid \Pi x{:}A\,.\,B \mid \lambda x{:}A\,.\,B \mid A\,M \\ \textit{Objects} & M & ::= & c \mid x \mid \lambda x{:}A\,.\,M \mid M\,N \end{array}$$

The binding operators Π and λ bind their first argument $x{:}A$ in the second argument position. It is conventional to identify terms that differ only in the names of bound variables, and the usual definitions of free and bound variables and substitution are used. The notation $[M_1,\ldots,M_k/x_1,\ldots,x_k]N$ denotes the term that results from the simultaneous substitution of $M_1,\ldots,M_k$ for free occurrences of $x_1,\ldots,x_k$ respectively in the term N, renaming bound variables as needed to avoid capture.

Families of the form $\Pi x{:}A.\ B$ are product types, which classify functions $\lambda x{:}A\,.\,M$ at the object level. When x does not occur free in B, the type $\Pi x{:}A.\ B$ is an ordinary function type; in this case it is customary to write it as $A \rightarrow B$. The Π operator quantifies over terms at the object level only. The application of one of these families to an object-level term x reduces to a type that depends on the value of x (if x appears free in B). Similarly, kinds of the form $\Pi x{:}A.\ K$ are product kinds, which classify dependent type families $\lambda x{:}A\,.\,B$. Again x ranges only over object-level terms, and when x is not free in K we write $A \rightarrow K$.

The crucial properties of the calculus are first, the decidability of the type system, which yields a proof checker for any deductive system properly encoded in it, and second, the existence of canonical forms, which permits the precise statement and proof of the correctness of an encoding. We discuss the correctness of the encodings of this chapter in Appendix B Rather than reproduce the calculus in full here, we refer the reader to [HHP93]; but in order to discuss the principles of the encodings we use, we describe some of its features.

The rules of the calculus assign a type A to an object-level term M in a context Γ and a signature Σ. Similarly they assign a kind K to a type A in a context Γ and a signature Σ. Signatures and contexts have the same structure and are used to maintain a typing environment, signatures being used to record the types and kinds of constants and contexts being used to record the types of variables.

$$\begin{array}{llll} \textit{Signatures} & \Sigma & ::= & <> \mid \Sigma, a{:}K \mid \Sigma, c{:}A \\ \textit{Contexts} & \Gamma & ::= & <> \mid \Gamma, c{:}A \end{array}$$

The calculus includes rules for deducing when a signature or context is valid, which guarantee that variables and constants are not "re-declared".

Encodings in LF of deductive systems that use collections of assumptions (for example, a type assignment system for a programming language, or systems of natural deduction) often represent these collections of assumptions in the meta-level context.

Hence it is important to keep in mind the properties of LF contexts stated in Theorem 2.3 of [HHP93]. We sketch them here at an intuitive level:

1. *Weakening:* a type or kind assignment that holds in a context Γ holds also in any valid extension of Γ.

2. *Strengthening:* a type or kind assignment α that holds in a context $\Gamma, x : U, \Gamma'$ holds also in Γ, Γ' provided that x is not free in Γ' or in α.

3. *Transitivity:* if the type assignment $M : A$ holds in context Γ, and the assignment α holds in context $\Gamma, x : A, \Gamma'$, then the assignment $[M/x]\alpha$ holds in the context $\Gamma, [M/x]\Gamma'$.

4. *Permutation:* if the assignment α holds in context $\Gamma, x : A, \Gamma', y : B, \Gamma''$, then it holds in the context $\Gamma, y : B, \Gamma', x : A, \Gamma''$, provided that x is not free in B or Γ' and B is a valid type or kind in Γ.

There are two important principles that are repeatedly used in LF representations of languages and deductive systems: *judgments as types* and *higher-order abstract syntax.* This section briefly introduces these principles; as they are encountered in the implementation, they are explained more fully.

The principle of judgments as types is crucial in encodings of deductive systems. A *judgment* is the relation established by a deductive system, for example, the truth of a logical formula or a typing assignment for a program expression. An LF encoding of a deductive system represents a proof as an object and a judgment as the type of its proof. That is, to encode a deductive system in LF, one declares a type family A to represent the judgment. The inference rules of the system are represented by declaring LF object-level constants that construct objects of that type family. A deduction in the system may then be represented as an LF object; it is a valid deduction if it can be given type A in the LF type system. Thus the principle of judgments-as-types gives rise to the corresponding principle that proof checking (in the encoded deductive system) is type checking (in the LF system). When A is a dependent type (has a Π-kind), LF type checking can guarantee some correctness properties of the inference system that would otherwise have to be proved in an external metatheory. The proof transformations of section 4 are an example; because we encode the judgment that a proposition A is provable as a dependent type, it is possible to implement proof transformations so that type checking guarantees that the transformed proof is not only valid, but still proves the same proposition A.

Judgments may be *basic, hypothetical,* or *schematic.* A basic judgment is one established by the logical system to be encoded, for example, that a formula is provable in first-order logic. We represent a basic judgment form by a type family indexed by the type(s) of the subject(s) of the judgment. A hypothetical judgment states that a judgment B is deducible from an assumption A. In LF this is encoded as a type $A \rightarrow B$; the encoding extends in an obvious way to encoding B deducible from multiple assumptions $A_1 \ldots A_n$ as $A_1 \rightarrow \ldots \rightarrow A_n \rightarrow B$. A schematic judgment that B holds for any term x of type A is encoded as a dependent function type $\Pi x{:}A\,.\,B$.

Higher-order abstract syntax eases the implementation of binding constructs in language definitions. Instead of explicitly naming variables and managing substitutions, the user can declare a binding construct as an LF constant that takes a functional argument and constructs a language expression. Terms are constructed by applying the constant to LF function objects. The usual rule of β-conversion is built in to the type theory, allowing substitution to be encoded as LF application. This technique extends beyond language definition to deductive systems. It is often appropriate and convenient to represent assumptions introduced during a deduction as bound variables at the meta-level, instead of managing them explicitly. As long as the object deduction system obeys the principles of weakening, strengthening, transitivity, and permutation described above for LF contexts, this kind of representation is faithful.

2.1.2 The Elf language

The core of the Elf syntax is a straightforward translation of LF syntax into a limited character set, with curly braces {} playing the role of Π and [] playing the role of λ. Type annotations may be omitted; we indicate this by enclosing them in brackets $\langle\rangle$. The symbol *id* stands for a variable or a constant.

Kinds	*kexp*	::=	`type` | {*id* ⟨:*fexp*⟩}*kexp*
Families	*fexp*	::=	*id* | {*id* ⟨:*fexp*⟩}*fexp* | [*id* ⟨:*fexp*⟩]*fexp* | *fexp oexp*
Objects	*oexp*	::=	*id* | [*id* ⟨:*fexp*⟩]*oexp* | *oexp oexp*

Elf also allows the use of arrow notation for a dependent type or kind `{x:A} M` where `x` is not free in `M`. For use in dynamic signatures, which are viewed as logic programs, there is also the backwards arrow notation `A <- B`, which stands for LF's $B \to A$. The arrow is right-associative (as usual), while the backwards arrow is left-associative. Thus there is an additional alternative for the category of Kinds and two more for the category of Families:

Kinds	*kexp*	::=	... | *fexp* `->` *kexp*
Families	*fexp*	::=	... | *fexp* `->` *fexp* | *fexp* `<-` *fexp*

A family or object expression may be annotated with its kind or type. This is useful in conjunction with Elf's type and term reconstruction since it can be used to name parameters that are normally left implicit. Thus there is one more alternative each for Families and Objects:

Families	*fexp*	::=	... | *fexp* : *kexp*
Objects	*oexp*	::=	... | *oexp* : *fexp*

Parentheses may be used freely to limit the scope of bound variables and change the associativity of arrows and type annotations. Any identifier may be used for a bound variable or constant. Identifiers free in a (top level) family or kind must be capitalized. In a type declaration these are implicitly Π-quantified. In a query they are treated as logic variables. Omitted type annotations and the types of free variables are supplied where possible by Elf's type reconstruction mechanism.

An Elf signature is a collection of kind and type declarations:

Declarations *decl* ::= *id*: *kexp*. | *id*: *fexp*.

A signature that is declared as *static* is not used for search and serves as a language definition. Kind and family declarations in these signatures establish syntactic categories, fitting the usual intuition about types. However, even when a signature is not used for search, its types may be thought of as judgments. This is seen in the representation of natural-deduction proofs of section 2.2.2, which supports simultaneously a treatment of proofs as objects and one of proofs as ways of establishing judgments.

A signature declared as *dynamic* may be viewed as a logic program and is used for search by the Elf interpreter. Types play the same role that formulas do in Prolog. A query is not a formula but a type, which may contain free variables. These are treated like the logic variables of Prolog: if the query succeeds, the substitution found by unification during type reconstruction or search for each free variable is displayed. Higher-order unification is used instead of first-order unification as in Prolog. To solve a query the interpreter attempts to construct an object of the given type, using the declarations in the dynamic signatures available. If the query succeeds, this object, which represents a proof of the goal query, is available as well as the substitutions for the free variables of the query. The proof object can itself be used in further goals. An example of this usage appears in the partially internalized proofs of correctness for the implementation in chapter 3. The availability of the proof object again manifests the dual nature of signatures: although a dynamic signature is usually thought of as a program for establishing judgments, it can also be thought of as a language definition, with its constituent declarations as constructors of terms in the language. These terms may themselves be objects of computation.

An important feature of dynamic signatures, corresponding to the use of higher-order abstract syntax in static signatures, is the way ordinary and dependent function types serve to introduce assumptions. When the interpreter encounters a goal that is a function type $A \rightarrow B$, A is added to its stock of rules as an assumption available during the search for a term of type B, causing subgoals that are unifiable with A to succeed in that search. A goal may also have dependent function type $\Pi x{:}A\,.\,B$, with x free in B. When attempting to solve such a goal the interpreter creates a new parameter x_0 of type A and a new subgoal B with x_0 substituted for free occurrences of x. The use of dependent types as subgoals is a common technique in Elf programs that implement deductive systems over terms represented as higher-order abstract syntax. In such a system, an inference rule that analyzes a binding construct frequently has a premise that depends on an assumption about the bound variable. For instance, one might infer a type for a λ-abstraction by adding a type assignment for its bound variable to a context and attempting to infer a type for its body. Dependently typed assumptions are typically used in encodings of inference rules of this form. The technique occurs frequently in the implementation described here; there is a more detailed description of it in section 2.3.1.

2.2 Logic

This section describes the encoding of natural-deduction style proofs for intuitionistic first-order predicate calculus. The encoding supports the manipulation of proofs as objects rather than proof search, and should be viewed as simply a set of constructor definitions defining the abstract syntax of a logical language and its proof system. Thus the Elf implementation consists of a static signature for each "language" – the language of propositions and the language of proofs. It does not provide a theorem prover for natural deduction.

We represent propositions of first-order predicate calculus as object-level LF terms. The language definition is a signature that declares the logical connectives and quantifiers as constructors. This contrasts with other treatments of constructive logic based on type theories, such as Nuprl and the Calculus of Constructions, in which propositions of the logic are identified with types of the type theory.

There are several reasons for separating the type of propositions from LF types. LF is a weak theory that does not include quantification over types. Thus it is not possible in LF to declare logical connectives as functions over terms of kind **Type**. Why then choose Elf for an implementation language? Proof transformations, program extraction, and related tasks are metaprogramming problems: they cannot usually be implemented for a type theory in that type theory. One standard approach to metaprogramming has been to use a programming language such as ML as a metalanguage for a type theory considered as an object logic. We want to implement proof transformations and other metaprograms in a flexible, declarative and verifiable way. As Basin and Constable [BC93] argue, these goals suggest another approach: the use of a type theory as a metalogical framework. Although, as they point out, the weakness of the LF type theory limits the metareasoning that can be fully internalized, the logic programming interpretation provided by Elf affords a flexible programming tool for expressing metatheory in a declarative way that also yields executable metaprograms. The availability of higher-order abstract syntax yields concise, quickly implementable encodings for many (though not all) deductive systems. This is a good setting for experimenting with variations on the object logic, the syntax and semantics of the programming language, and program extraction, without the need to modify the syntax or semantics of the underlying type theory. We in fact exploit this by exploring two definitions of program extraction. While one must expend the effort to explicitly define these systems, this effort is typically small, especially when higher-order abstract syntax can be used. The reward is flexibility in choices such as the definition of program extraction, and the syntax and semantics of the programming language used for extraction.

2.2.1 Logical language

Our encoding of first-order constructive logic follows the method described in Harper et al. [HHP93]. To simplify the exposition, we show the representation for a logic with only a single sort of individuals, interpreted as natural numbers. Later in the presentation we show how to modify the definitions to represent a many-sorted logic.

An abstract syntax for the logic is:

$$\begin{array}{llll} \textit{Individuals} & t & ::= & x \mid \mathrm{zero} \mid \mathrm{succ}\ t \\ \textit{Propositions} & A & ::= & \top \mid \bot \mid A_1 \wedge A_2 \mid A_1 \vee A_2 \mid A_1 \supset A_2 \mid \neg A \\ & & & \mid \forall x . A \mid \exists x . A \mid t_1 = t_2 \end{array}$$

To encode this syntax in LF we first define two new types, o for the type of propositions and i

for the type of individuals:

$$\begin{aligned} o&: \text{Type} \\ i&: \text{Type} \end{aligned}$$

Individuals are encoded by declaring two constants for representing zero and the successor function:

$$\begin{aligned} \text{zero}&: \ i \\ \text{succ}&: \ i \rightarrow i \end{aligned}$$

There is no declaration corresponding to individual variables; they are represented as LF bound variables through the use of higher-order abstract syntax.

Connectives are encoded in a straightforward way as syntactic constructors. Binary constructors are curried since there are no finite products in LF.

$$\begin{aligned} \top&: \ o \\ \bot&: \ o \\ \neg&: \ o \rightarrow o \\ \wedge&: \ o \rightarrow o \rightarrow o \\ \vee&: \ o \rightarrow o \rightarrow o \\ \supset&: \ o \rightarrow o \rightarrow o \end{aligned}$$

The equality predicate is encoded with a curried function type as well:

$$=: \ i \rightarrow i \rightarrow o$$

The encoding of quantifiers is a simple use of higher-order abstract syntax. Each quantifier is encoded as a constructor that takes, not a proposition, but a function from an individual to a proposition. The binding properties of the quantifiers are expressed by LF λ-binding. For example, in the proposition $\forall x \,.\, x = x$ the two occurrences of x in the equality are bound by the quantifier. The LF representation is $\forall(\lambda x{:}i \,.\, x = x)$, an application of the constructor $\forall$ to an LF object of functional type. Thus the schematic nature of a universally or existentially quantified proposition is captured by the LF meta-language and does not have to be explicitly managed. Only the semantic distinction – the universal vs. existential nature of the proposition – is left to be expressed by the encodings of inference rules, reductions, etc.

$$\begin{aligned} \forall&: \ (i \rightarrow o) \rightarrow o \\ \exists&: \ (i \rightarrow o) \rightarrow o \end{aligned}$$

The Elf signature (Figure 2.1) corresponding to the LF declarations presents no additional complications. The differences are due purely to the concrete syntax of Elf.

2.2.2 Natural deduction

In our encoding of proofs, again following Harper et al. [HHP93], we use the dependent types of LF to ensure that proof checking is LF type checking – even though propositions are object-level terms,

```
o : type.
i : type.

zero : i.
succ : i -> i.
eq : i -> i -> o.

true : o.
false : o.
not : o -> o.
and : o -> o -> o.
or : o -> o -> o.
implies : o -> o -> o.
forall : (i -> o) -> o.
exists : (i -> o) -> o.
```

Figure 2.1: Elf encoding of first-order logic

not LF types. Higher-order abstract syntax supports the representation of the introduction and discharge of assumptions, as well as the side conditions restricting the free occurrences of variables in assumptions.

As is customary, we use a tree-like notation in our presentations of natural deduction proofs. Assumptions are at the leaves, and a formula follows from zero or more formulas immediately above it by one of the inference rules of the system. Assumptions may be *discharged* or *closed* by an inference (e.g., implication introduction); an undischarged assumption is often called *open*. We say a formula depends on the open assumptions above it in the proof tree. The last formula (at the root of the tree) of a proof is a theorem if it does not depend on any open assumptions. There is no distinguished set of axioms, though we sometimes refer to an inference rule with no premises as an axiom.

Figure 2.2 gives the inference rules for natural-deduction style proofs in an intuitionistic first-order predicate logic. The discharge of an assumption A by an inference rule is indicated by placing an annotated bar over A:
$\overline{A}^{p}$ The corresponding rule is indicated by a matching superscript (e.g. $\supset\text{I}^p$). We use either numbers or letters for superscripts, depending on the context. Letters are useful in the context of program extraction, where a discharged assumption corresponds to a bound variable in the extracted program. Substitution is notated as follows: $[t/x]A$ denotes the substitution of t for free occurrences of x in A. We use this notation freely in subsequent language definitions, assuming standard definitions of bound and free occurrences, and the ability to rename variables whenever necessary. The usual conditions on free occurrences of variables apply to the quantifier rules annotated with an asterisk. In a proof of $\forall x . A$ ending in the use of the rule $\forall\text{I}$, the variable x cannot occur free in any undischarged assumption. In a proof of C by $\exists\text{E}$ with $\exists x . A$ as major premise, the variable x cannot occur free in C or in any undischarged assumption on which the

$$\frac{A \qquad B}{A \wedge B}\,\wedge\mathrm{I}$$

$$\frac{A \wedge B}{A}\,\wedge\mathrm{E}_L \qquad\qquad \frac{A \wedge B}{B}\,\wedge\mathrm{E}_R$$

$$\frac{A}{A \vee B}\,\vee\mathrm{I}_L \qquad\qquad \frac{B}{A \vee B}\,\vee\mathrm{I}_R$$

$$\frac{A \vee B \qquad \begin{array}{c}\overline{A}^{\,p}\\ \vdots \\ C\end{array} \qquad \begin{array}{c}\overline{B}^{\,p}\\ \vdots \\ C\end{array}}{C}\,\vee\mathrm{E}^p$$

$$\frac{\begin{array}{c}\overline{A}^{\,p}\\ \vdots \\ B\end{array}}{A \supset B}\,\supset\mathrm{I}^p \qquad\qquad \frac{A \supset B \qquad A}{B}\,\supset\mathrm{E}$$

$$\frac{A}{\forall x\,.\,A}\,\forall\mathrm{I}\,* \qquad\qquad \frac{\forall x\,.\,A}{[t/x]A}\,\forall\mathrm{E}$$

$$\frac{[t/x]A}{\exists x\,.\,A}\,\exists\mathrm{I} \qquad\qquad \frac{\exists x\,.\,A \qquad \begin{array}{c}\overline{A}^{\,p}\\ \vdots \\ C\end{array}}{C}\,\exists\mathrm{E}^p\,*$$

$$\frac{\begin{array}{c}\overline{A}^{\,p}\\ \vdots \\ \bot\end{array}}{\neg A}\,\neg\mathrm{I}^p \qquad\qquad \frac{A \qquad \neg A}{\bot}\,\neg\mathrm{E}$$

$$\frac{}{\top}\,\top\mathrm{I} \qquad\qquad \frac{\bot}{C}\,\bot\mathrm{E}$$

Figure 2.2: Natural deduction rules for first-order logic

$$\frac{}{t = t}\,{=}\mathrm{R} \qquad\qquad \frac{t_2 = t_1}{t_1 = t_2}\,{=}\mathrm{Y}$$

$$\frac{t_1 = t_2 \qquad t_2 = t_3}{t_1 = t_3}\,{=}\mathrm{T} \qquad\qquad \frac{t_1 = t_2}{f(t_1) = f(t_2)}\,{=}\mathrm{U}$$

$$\frac{}{\neg\mathrm{succ}\ t = \mathrm{zero}}\,\mathrm{AXO} \qquad\qquad \frac{\mathrm{succ}\ t_1 = \mathrm{succ}\ t_2}{t_1 = t_2}\,\mathrm{AXS}$$

$$\frac{[\mathrm{zero}/x]A \qquad \begin{array}{c}\overline{A}^{\,p}\\ \vdots \\ [\mathrm{succ}\ x/x]A\end{array}}{\forall x\,.\,A}\,\mathrm{IND}^p$$

Figure 2.3: Inference rules for equality and arithmetic

subproof of C depends.

Figure 2.3 presents inference rules for interpreting the individuals of the logic as natural numbers. For simplicity in the presentation we defer discussion of other theories to later sections.

Note that the treatment of assumptions and the side conditions on variable occurrences means that the inference rules $\supset$I, $\forall$I, $\vee$E, $\exists$E, and IND are binding constructs. Consider implication introduction:

$$\frac{\begin{array}{c}\overline{A}^{\,p}\\ \vdots \\ B\end{array}}{A \supset B}\,{\supset}\mathrm{I}^p$$

Here p is a name for a hypothetical proof, as the bound variable in a λ-abstraction is a name for a hypothetical value. The scope of p is limited to the subproof ending in the application of $\supset$I labelled p; no other part of the proof tree can include assumptions of A labelled p. Many of the operations on proofs needed in the implementation, including program extraction and proof transformation, use the substitution of proofs for assumptions and terms for quantified variables in a way that conforms to LF β-reduction. Such usage motivates the higher-order abstract syntax approach to the encoding of proofs.

A proof of a minimal correctness criterion for this kind of encoding of natural deduction is given by Harper et al. [HHP93].

Figure 2.4 shows the LF encoding of the logical inference rules; Figure 2.5 shows the encoding

pf:	$o \to \mathrm{Type}$
⊤I:	$\mathsf{pf}\,\top$
⊥E:	$\Pi C{:}o.\ \mathsf{pf}\,\bot \to \mathsf{pf}\,C$
∧I:	$\Pi A{:}o.\Pi B{:}o.\ \mathsf{pf}\,A \to \mathsf{pf}\,B \to \mathsf{pf}\,A \wedge B$
$\wedge E_L$:	$\Pi A{:}o.\Pi B{:}o.\ \mathsf{pf}(A \wedge B) \to \mathsf{pf}\,A$
$\wedge E_R$:	$\Pi A{:}o.\Pi B{:}o.\ \mathsf{pf}(A \wedge B) \to \mathsf{pf}\,B$
$\vee I_L$:	$\Pi A{:}o.\Pi B{:}o.\ \mathsf{pf}\,A \to \mathsf{pf}(A \vee B)$
∨E:	$\Pi A{:}o.\Pi B{:}o.\ \mathsf{pf}(A \vee B) \to (\mathsf{pf}\,A \to \mathsf{pf}\,C) \to (\mathsf{pf}\,B \to \mathsf{pf}\,C) \to \mathsf{pf}\,C$
$\vee I_R$:	$\Pi A{:}o.\Pi B{:}o.\ \mathsf{pf}\,B \to \mathsf{pf}(A \vee B)$
⊃I:	$\Pi A{:}o.\Pi B{:}o.(\mathsf{pf}\,A \to \mathsf{pf}\,B) \to \mathsf{pf}(A \supset B)$
⊃E:	$\Pi A{:}o.\Pi B{:}o.\ \mathsf{pf}(A \supset B) \to \mathsf{pf}\,A \to \mathsf{pf}\,B$
∀I:	$\Pi A{:}(i \to o).(\Pi x{:}i.\ \mathsf{pf}(Ax)) \to \mathsf{pf}(\forall A)$
∀E:	$\Pi A{:}(i \to o).\Pi t{:}i.\ \mathsf{pf}(\forall A) \to \mathsf{pf}\,A(t)$
∃I:	$\Pi A{:}(i \to o).\Pi t{:}i.\ \mathsf{pf}\,At \to \mathsf{pf}(\exists A)$
∃E:	$\Pi A{:}i \to o.\ \mathsf{pf}\,\exists A \to (\Pi x{:}i.\ \mathsf{pf}\,Ax \to \mathsf{pf}\,C) \to \mathsf{pf}\,C$

Figure 2.4: LF encoding of logical inference rules

=REFL:	$\Pi t{:}i.\ \mathsf{pf}(t = t)$
=SYM:	$\Pi t_1{:}i.\Pi t_2{:}i.\ \mathsf{pf}(t_2 = t_1) \to \mathsf{pf}(t_1 = t_2)$
=TRANS:	$\Pi t_1{:}i.\Pi t_2{:}i.\Pi t_3{:}i.\ \mathsf{pf}(t_1 = t_2) \to \mathsf{pf}(t_2 = t_3) \to \mathsf{pf}(t_1 = t_3)$
=SUBST:	$\Pi f{:}i \to i.\Pi t_1{:}i.\Pi t_2{:}i.\Pi t_3{:}i.\ \mathsf{pf}(t_1 = t_2) \to \mathsf{pf}(f(t_1) = f(t_2))$
AX0:	$\forall(\lambda x.\neg \mathrm{succ}\ x = \mathrm{zero})$
AXSUCC:	$\forall(\lambda x.\forall\lambda y.\mathrm{succ}\ x = \mathrm{succ}\ y)$
IND:	$\Pi A{:}i \to o.\ \mathsf{pf}\,A\ \mathrm{zero} \to (\Pi x{:}i.\ \mathsf{pf}\,Ax \to \mathsf{pf}\,A(\mathrm{succ}\ x)) \to \mathsf{pf}\,\forall A$

Figure 2.5: LF encoding of inference rules for equality and arithmetic

of rules for equality and arithmetic. The encoding is based on the declaration of a dependent type constructor (family):

$$\mathsf{pf}: \ o \to \text{Type}$$

The dependent type provides an association between a proof and the proposition it proves, encoded within the LF type of the term representing the proof. This association permits the encoding of inference rules to enforce the constraint that any well-typed proof term is a valid proof, i.e., to maintain the principle that proof checking is LF type checking.

For example, the rule of conjunction elimination on the left is encoded as follows.

$$\wedge \mathrm{E}_L: \ \Pi A{:}o\,.\,\Pi B{:}o\,.\ \mathsf{pf}(A \wedge B) \to \mathsf{pf}\, A$$

This defines the constructor $\wedge\mathrm{E}_L$ as a function that takes as input propositions A and B, and a proof of $A \wedge B$. The constructed term has type $\mathsf{pf}\, A$, i.e., it represents a proof of A. Because of the dependent type, LF type checking prevents the construction of an incorrect proof. For instance, there is no way to represent the application of $\wedge\mathrm{E}_L$ to a proof of $\top$ by $\top\mathrm{I}$, since it has type $\mathsf{pf}\,\top$.

The other inference rules not involving bound variables or assumptions are encoded similarly.

A simple use of higher-order abstract syntax occurs in the encoding of the rules $\supset$I, where it is used to express the discharge of an assumption, and $\forall$I, where it is used to express the restriction on free occurrences of the universally quantified variable.

The rule $\supset$I

$$\cfrac{\begin{array}{c}\overline{A}^{\,p}\\ \vdots \\ B\end{array}}{A \supset B}\,\supset\mathrm{I}^p$$

is encoded as follows.

$$\supset\mathrm{I}: \ \Pi A{:}o\,.\,\Pi B{:}o\,.(\mathsf{pf}\, A \to \mathsf{pf}\, B) \to \mathsf{pf}(A \supset B)$$

This declaration defines the constructor $\supset$I as a function that takes propositions A and B, and a function from proofs of A to proofs of B, and constructs a proof of $A \supset B$.

A small example proof shows how functional abstraction in LF models the use of assumptions. Consider the following deduction.

$$\cfrac{\overline{\text{zero} = \text{succ zero}}^{\,p}}{\text{zero} = \text{succ zero} \supset \text{zero} = \text{succ zero}}\,\supset\mathrm{I}^p$$

Its LF representation is the closed term:

$$\supset\mathrm{I}\,(\text{zero} = \text{succ zero})(\text{zero} = \text{succ zero})(\lambda p{:}\mathsf{pf}(\text{zero} = \text{succ zero})\,.\,p)$$

This term has type $\mathsf{pf}(\text{zero} = \text{succ zero} \supset \text{zero} = \text{succ zero})$.

On the other hand, the following deduction contains an undischarged assumption and cannot be represented by a closed LF term:

$$\dfrac{\dfrac{\dfrac{}{\text{zero} = \text{succ zero}}\,p}{\text{zero} = \text{succ zero} \supset \text{zero} = \text{succ zero}}\,{\supset}\mathrm{I}^{p} \qquad \text{zero} = \text{succ zero}}{(\text{zero} = \text{succ zero} \supset \text{zero} = \text{succ zero}) \wedge \text{zero} = \text{succ zero}}\,{\wedge}\mathrm{I}$$

In the corresponding LF term:

$$\begin{aligned}&{\wedge}\mathrm{I}\ ((\text{zero} = \text{succ zero}) \supset (\text{zero} = \text{succ zero}))\,(\text{zero} = \text{succ zero})\\&\quad ({\supset}\mathrm{I}\ (\text{zero} = \text{succ zero})\,(\text{zero} = \text{succ zero})\,(\lambda p{:}\,\mathsf{pf}(\text{zero} = \text{succ zero})\,.\,p))\ p\end{aligned}$$

the second occurrence of p is free, representing the open assumption.

Similarly, the encoding of ∀I models the schematic nature of the premise and enforces the stipulation that there be no free occurrences of the quantified variable in any open assumption. The rule is encoded as follows:

$$\forall\mathrm{I}: \ \ \Pi A{:}(i \to o)\,.\,(\Pi x{:}i\,.\,\mathsf{pf}(Ax)) \to \mathsf{pf}(\forall A)$$

The constructor ∀I has two functional inputs. The first, of type $i \to o$, represents a proposition that is schematic in an individual x. The second, of (dependent) type $(\Pi x{:}i\,.\,\mathsf{pf}(Ax))$, represents a schematic proof. The constructed term represents a proof (in which the ∀I rule acts as binder) of a universal quantification over x (in which ∀ acts as a binder).

As an example, we examine the representation of the following proof:

$$\dfrac{\dfrac{}{x = x}\,{=}\mathrm{R}}{\forall x\,.\,x = x}\,\forall\mathrm{I}$$

This is encoded as:

$$\forall\mathrm{I}(\lambda x{:}i\,.\,x = x)(\lambda x{:}i\,.\ {=}\mathrm{REFL}\,x)$$

Now consider the following "deduction", which is invalid because x occurs free in an undischarged assumption.

$$\dfrac{x = \text{zero}}{\forall x\,.\,x = \text{zero}}\,\forall\mathrm{I}$$

This cannot be represented. An attempt to represent it might begin:

$$\forall\mathrm{I}(\lambda x{:}i\,.\,x = \text{zero})(\lambda x{:}i\,.\,\ldots)$$

But there is no constructor that, given a term x of type i, constructs a proof of $x = \text{zero}$.

The representation of the other inference rules involving the discharge of assumptions and the binding of variables follows the same principles.

The expression of these LF declarations in the Elf language (Figure 2.6) is straightforward, except for the use of implicit quantification, which is used to reduce the verbosity of the "pure" LF declarations.

The declaration of the type family of proofs is identical to the LF declaration, but we use the turnstile-like notation "|-" in place of the constructor **pf**.

```
|- : o -> type.

truei : |- true.

falsee : {C:o} |- false -> |- C.

andi : |- A -> |- B -> |- (and A B).
andel : |- (and A B) -> |- A.
ander : |- (and A B) -> |- B.

oril : {B:o} |- A -> |- (or A B).
orir : {A:o} |- B -> |- (or A B).
ore : |- (or A B) -> (|- A -> |- C) -> (|- B -> |- C) -> |- C.

impliesi : (|- A -> |- B) -> |- (implies A B).
impliese : |- (implies A B) -> |- A -> |- B.

noti : (|- A -> |- false) -> |- (not A).
note : |- (not A) ->  |- A  -> |- false.

foralli : ({x:i} |- (A x)) -> |- (forall A).
foralle : {T:i} |- (forall A) -> |- (A T).

existsi : {A:i -> o} {T:i} |- (A T) -> |- (exists A).
existse : |- (exists A) -> ({x:i} |- (A x) -> |- C) -> |- C.

eq_refl : {X:i} |- (eq X X).
eq_sym : |- (eq X Y) -> |- (eq Y X).
eq_trans : |- (eq X Y) -> |- (eq Y Z) -> |- (eq X Z).
eq_subst : |- (eq X Y) -> |- (eq (succ X) (succ Y)).

ax_zero : |- (forall [x] (not (eq (succ x) zero))).
ax_succ : |- (forall [x] (forall [y] (implies (eq (succ x) (succ y))
                                               (eq x y)))).

ind : {A:i -> o} |- (A zero) -> ({x:i} |- (A x) -> |- (A (succ x)))
        -> |- (forall A).
```

Figure 2.6: Elf encoding of logical, equality, and arithmetic inference rules

```
|- : o -> type.
```

Because of Elf's term and type reconstruction facility, some arguments of the LF declarations can be omitted in the Elf encoding, and reconstructed during type checking. For example, the LF declaration for conjunction introduction is

$$\wedge\mathrm{I}: \ \Pi A{:}o.\Pi B{:}o.\ \mathsf{pf}\,A \to \mathsf{pf}\,B \to \mathsf{pf}\,A\ \wedge\ B.$$

But in Elf this can be written

```
andi : |- A -> |- B -> |- (and A B).
```

`A` and `B` are implicitly quantified in this declaration. When type checking the declaration Elf's type reconstruction can infer that `A` and `B` must have type `o` since `|-` takes a term of that type. When constructing a term by this rule the two propositions are not supplied explicitly, since Elf reconstructs them from their proofs.

On the other hand, consider the following declaration:

```
falsee : {C:o} |- false -> |- C.
```

If `C` were not explicitly quantified, type reconstruction could still produce the correct type for the declaration. But to use the rule in the construction of a proof, it is sometimes necessary to explicitly supply `C` since it cannot always be inferred from the input proof. Therefore we include the explicit quantification in the declaration so that an actual parameter for `C` is accepted by the Elf front end when the user inputs a proof. An alternative method is to make the parameter implicit:

```
falsee : |- false -> |- C.
```

Then if `C` cannot be inferred from the context, it can be specified in a type annotation, e.g.

```
impliesi [p: |- false] (falsee: |- (eq zero (succ zero)))
```

2.3 Program extraction

To implement the extraction of programs from proofs, we first define a simple language of functional programs. The syntax of the language is implemented in Elf according to the same principles used in the definition of predicate calculus in section 2.2.1. A typing discipline and operational semantics for the language are defined in the form of deductive systems encoded as dynamic Elf signatures. As with the encoding of logic, the representation of programs is external to the underlying logical framework. Our encodings are in the style of those given by Michaylov and Pfenning for a fragment of ML in [MP91], which extend the higher-order representations developed in λProlog by Hannan and Miller in [HM89], [Han91].

2.3.1 Programming language

Because we are explicitly defining a language for extracted programs, external to the LF type theory, there are syntactic and semantic choices open to us. The language presented here is a simple extension of a λ-calculus with primitive recursion, pairs, and disjoint union, with a call-by-value semantics. The language constructs are chosen to correspond closely to the inference rules of the logic. Extraction into more expressive languages is not difficult to implement along similar lines, and we have done so for a miniML-like language.

Syntax of expressions and types

In the presentation of extraction and its meta-theory we use a Nuprl-like syntax for programs [C+86] in order to emphasize the close relation to the logic. (The semantics of the language, however, is not the Nuprl semantics.) A BNF grammar for the language follows; it consists of the single syntactic category of expressions e.

$$
\begin{array}{rcll}
e & ::= & x \mid & \textit{Variables} \\
 & & 0 \mid \mathrm{s}(e) \mid & \textit{Natural numbers} \\
 & & \langle e_1, e_2 \rangle \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \mid \mathbf{spread}(e_1;\ x,y\,.\,e_2) \mid & \textit{Pairs} \\
 & & \mathbf{inl}(e) \mid \mathbf{inr}(e) \mid \mathbf{decide}(e_1;\ x\,.\,e_2;\ x\,.\,e_3) \mid & \textit{Disjoint union} \\
 & & \mathbf{lam}\, x\,.\,e \mid \mathbf{nat\_ind}(e_1;\ x,y\,.\,e_2) \mid \mathbf{app}(e_1, e_2) \mid & \textit{Functions} \\
 & & () \mid & \textit{Unity} \\
 & & \mathbf{any}(e) \mid \mathbf{neg} & \textit{Error} \\
 & & \mathbf{axiom} & \textit{Self-realizors}
\end{array}
$$

(Later in the presentation we add more language constructs as we deal with different theories.)

If we wish only to extract programs from proofs and execute them, there is no need to consider a type structure for this language. However, we also want to be able to prove (and to do so partially within Elf) some metatheorems about the correctness of extraction and program execution. For instance, it is evident from the interpretation of constructive logic described in Section 1.1 that the type corresponding to a proposition in the object logic of the form $A \wedge B$ should be a product. Moreover, the evaluation of an extracted term should preserve this type. In order to reason about such properties we introduce a system of simple types for the language:

$$\textit{Types}\ \ \tau\ ::=\ \mathsf{nat} \mid \tau_1 \times \tau_2 \mid \tau_1 | \tau_2 \mid \tau_1 \Rightarrow \tau_2 \mid \mathsf{unit} \mid \mathsf{void} \mid \mathsf{atom}$$

These types should be carefully distinguished from the types of the LF calculus; we sometimes refer to them as *object types*.

The type system, the operational semantics of the language, and their role in representing the computational content of proofs are defined precisely in the form of deductive systems given in the rest of this chapter. Here we informally sketch the main outlines of these ideas.

The program extraction process expresses the computational content of a proof in the form of a program, losing some of the purely logical content in the translation. (Ideally, all of the purely logical content is removed; we examine some of the issues this raises in section 2.3.2.) The propositions-as-types principle implies a similar process of type extraction: to each logical proposition there corresponds an object type of the programming language.

The type **unit** is inhabited by one value, (), and corresponds to logical truth. Thus the value () is extracted from the rule of truth introduction.

A pair type may correspond to a conjunction or an existential. When $\tau_1 \times \tau_2$ corresponds to $A \wedge B$, τ_1 corresponds to A and τ_2 to B. When it corresponds to $\exists x \,.\, A$, τ_1 corresponds to the sort of x (in the object logic presented so far, this must be the sort of natural numbers i, but later we will extend the object logic to a many-sorted logic), and τ_2 corresponds to A. Thus pair formation is extracted from the introduction of conjunction or existential quantification. The usual projections, **fst** and **snd**, are extracted from conjunction eliminations. The destructor **spread** is extracted for existential elimination. The evaluation of **spread**(e_1; $x, y \,.\, e_2$) proceeds as follows. The expression e_1 is evaluated to obtain a pair $\langle\, v_1, v_2 \,\rangle$. Then v_1 is substituted for x and v_2 for y during the evaluation of e_2. The projections are of course redundant, as they are definable from **spread**, but the current version of Elf does not include a definition facility, so they are included in the language for the sake of program readability and the simplicity of the metatheoretical correctness proofs.

A disjoint union type $\tau_1 \mid \tau_2$ corresponds to a disjunction $A \vee B$, where τ_1 corresponds to A and τ_2 to B. The left and right injections **inl** and **inr** are extracted from the disjunction introduction rules, and the **decide** destructor from disjunction elimination.

A function type may correspond to an implication or a universal quantification. When $\tau_1 \Rightarrow \tau_2$ corresponds to $A \supset B$, τ_1 corresponds to A and τ_2 to B. When it corresponds to $\forall x \,.\, A$, τ_1 corresponds to the sort of x (the same considerations apply here as for $\exists x \,.\, A$) and τ_2 to B. We use **lam** to express functional abstraction and **app** to express application in order to distinguish them from LF functional abstraction and application. A **lam**-abstraction is extracted from the introduction rule for implication or universal quantification, and an application from the corresponding eliminations. The **nat_ind** constructor permits the definition of primitive recursive functions and is extracted from inductive proofs.

The type **void** corresponds to absurdity (falsehood). For a negation $\neg A$ there is a corresponding type $\tau \Rightarrow$ **void** where τ corresponds to A; we extract a function of this type from the rule of negation introduction. There is a redundancy in our object logic definition in that negation is definable in terms of implication and absurdity. Like **fst** and **snd**, it is included for notational convenience in the absence of a definition facility in the Elf language. The extraction process collapses the two notations at the level of types: the term extracted from a proof of $\neg A$ has the same type as the term extracted from a proof of $A \supset \bot$. Such proofs can arise only from the use of negative assumptions or axioms. Our simple system contains only one negative axiom schema AX0. For an instance of the schema the extraction procedure extracts **neg** with object type $\tau \Rightarrow$ **void** (for any object type τ). The constructor **any** is used in extractions from the rule of falsehood elimination. This rule allows an arbitrary proposition C to be inferred from a proof of absurdity. Correspondingly, the object type inference system allows an arbitrary type to be inferred for a term of the form **any**(e) if e has type **void**. Since there is no closed proof of absurdity in a consistent logic, evaluation of a term of the form **any**(e) represents an error. The operational semantics models this error by a (finite) failure to evaluate. There is no evaluation rule for the constructor **any**, and an Elf query of the form `?- eval (any M) V` will terminate with failure. Operationally this failure to evaluate is distinct from the nontermination of a query like `?- eval (app (lam [x] app x x) (lam [x] app x x)) V`. But declaratively both failures are equivalent: there is no evaluation deduction that inhabits the type of either query.

The type **atom** contains one element **axiom** and corresponds to equality assertions. This is a somewhat arbitrary choice as there is no computational content to a proof of equality; indeed in Kleene-style definitions of realizability in the domain of Heyting arithmetic any number will serve as a realizor of an equality. The types **atom** and **unit** are isomorphic and will be collapsed together when we define a more efficient version of extraction.

As there are no new principles involved in the encoding of the syntax of programs and types in LF, we show only its implementation in Elf (figure 2.7). Again, constructors with multiple arguments are curried, and higher-order abstract syntax is used for the binding constructors **lam**, **nat_ind**, **spread**, and **decide**.

We represent the types of our programming language explicitly as LF object-level terms. Thus we define a syntactic category `tp` of program types as well as the syntactic category `term` of program terms. Types are associated to terms by the type inference system presented below. The representation of types by LF objects is convenient for codifying the correspondence between propositions of the object logic and types of programs in terms of a type extraction system.

Operational semantics

We give a call-by-value operational semantics for the language as a set of inference rules (figure 2.8) defining a judgment $e \hookrightarrow v$ (e evaluates to v). This is a *natural semantics* in the style of Kahn [Kah87]. The choice of call-by-value as opposed to call-by-name is an arbitrary one, although it does simplify translation into ML, which we have done for some extracted programs.

We show the Elf implementation of the semantics in figure 2.9. This encoding is the first instance of an Elf signature used for search, i.e., a *program* as opposed to a language definition. We think of the family `eval` primarily as representing a judgment rather than a syntactic category. However, the signature also represents the syntactic category of evaluation deductions, and the internalized metatheory of Chapter 3 exploits this representation. For the purposes of introducing simple programming in Elf it is easier to first view the signature as a Prolog-like program. A declaration such as

```
ev_s : eval (s M) (s V) <- eval M V.
```

can be read as a Prolog-like rule, with the backwards arrow `<-` playing the role of Prolog's `:-` syntax. In the LF language, this declaration is equivalent to

$$\mathsf{ev\_s}\colon\ \Pi M{:}\mathsf{term}\,.\,\Pi V{:}\mathsf{term}\,.\,\mathsf{eval}\ M\ V \rightarrow \mathsf{eval}\,(\mathsf{s}(M))\,(\mathsf{s}(V))$$

This declares **ev_s** as a constructor in the same way as the declarations we have dealt with heretofore. But operationally, it is a constructor available to the Elf search mechanism as it constructs a proof of a *judgment*, and so it acts as a logic-programming rule.

Examination of the clauses for applications reveals the inefficiency of this implementation due to the backtracking and consequent repeated evaluation of `M` when evaluating `app M N`. Nevertheless the signature does define an operational interpreter and its structure is suitable for the partially internalized metatheory of chapter 3.

```
tp : type.
term : type.

arrow : tp -> tp -> tp.

app : term -> term -> term.
lam : (term -> term) -> term.

unit : tp.
unity : term.

nat : tp.
0 : term.
s : term -> term.
nat_ind : term -> (term -> term -> term) -> term.

* : tp -> tp -> tp.
pair : term -> term -> term.
fst : term -> term.
snd : term -> term.
spread : term -> (term -> term -> term) -> term.

| : tp -> tp -> tp.
inl : term -> term.
inr : term -> term.
decide : term -> (term -> term) -> (term -> term) -> term.

void : tp.
any : term -> term.
neg : term.

atom : tp.
axiom : term.
```

Figure 2.7: Elf encoding of the syntax of program expressions and types

$$\frac{}{0 \hookrightarrow 0}\,\textsf{ev-0} \qquad\qquad \frac{e \hookrightarrow v}{\mathsf{s}(e) \hookrightarrow \mathsf{s}(v)}\,\textsf{ev-s}$$

$$\frac{e_1 \hookrightarrow v_1 \qquad e_2 \hookrightarrow v_2}{\langle\, e_1, e_2 \,\rangle \hookrightarrow \langle\, v_1, v_2 \,\rangle}\,\textsf{ev-pair} \qquad \frac{e_1 \hookrightarrow \langle\, v_1, v_2 \,\rangle \qquad [v_1/x][v_2/y]e_2 \hookrightarrow v}{\mathbf{spread}(e_1;\ x, y\,.\,e_2) \hookrightarrow v}\,\textsf{ev-spread}$$

$$\frac{e \hookrightarrow \langle\, v_1, v_2 \,\rangle}{\mathbf{fst}(e) \hookrightarrow v_1}\,\textsf{ev-fst} \qquad\qquad \frac{e \hookrightarrow \langle\, v_1, v_2 \,\rangle}{\mathbf{snd}(e) \hookrightarrow v_2}\,\textsf{ev-snd}$$

$$\frac{e \hookrightarrow v}{\mathbf{inl}(e) \hookrightarrow \mathbf{inl}(v)}\,\textsf{ev-inl} \qquad\qquad \frac{e \hookrightarrow v}{\mathbf{inr}(e) \hookrightarrow \mathbf{inr}(v)}\,\textsf{ev-inr}$$

$$\frac{e_1 \hookrightarrow \mathbf{inl}(v_1) \qquad [v_1/x]e_2 \hookrightarrow v}{\mathbf{decide}(e_1;\ x\,.\,e_2;\ x\,.\,e_3) \hookrightarrow v}\,\textsf{ev-dec-l} \qquad \frac{e_1 \hookrightarrow \mathbf{inr}(v_1) \qquad [v_1/x]e_3 \hookrightarrow v}{\mathbf{decide}(e_1;\ x\,.\,e_2;\ x\,.\,e_3) \hookrightarrow v}\,\textsf{ev-dec-r}$$

$$\frac{}{\mathbf{lam}\, x\,.\,e \hookrightarrow \mathbf{lam}\, x\,.\,e}\,\textsf{ev-lam} \qquad \frac{e_1 \hookrightarrow \mathbf{lam}\, x\,.\,e \qquad e_2 \hookrightarrow v_2 \qquad [v_2/x]e \hookrightarrow v}{\mathbf{app}(e_1, e_2) \hookrightarrow v}\,\textsf{ev-app-lam}$$

$$\frac{}{\mathbf{nat\_ind}(e_1;\ x, y\,.\,e_2) \hookrightarrow \mathbf{nat\_ind}(e_1;\ x, y\,.\,e_2)}\,\textsf{ev-pr}$$

$$\frac{e_1 \hookrightarrow \mathbf{nat\_ind}(e_z;\ x, y\,.\,e_s) \qquad e_2 \hookrightarrow 0 \qquad e_z \hookrightarrow v}{\mathbf{app}(e_1, e_2) \hookrightarrow v}\,\textsf{ev-pr-z}$$

$$\frac{e_1 \hookrightarrow \mathbf{nat\_ind}(e_z;\ x, y\,.\,e_s) \quad e_2 \hookrightarrow \mathsf{s}(v_2) \quad \mathbf{app}((\mathbf{nat\_ind}(e_z;\ x, y\,.\,e_s)), v_2) \hookrightarrow v_3 \quad [v_2/x][v_3/y]e_s \hookrightarrow v}{\mathbf{app}(e_1, e_2) \hookrightarrow v}\,\textsf{ev-pr-s}$$

$$\frac{}{() \hookrightarrow ()}\,\textsf{ev-unit} \qquad \frac{}{\mathbf{axiom} \hookrightarrow \mathbf{axiom}}\,\textsf{ev-ax} \qquad \frac{}{\mathbf{neg} \hookrightarrow \mathbf{neg}}\,\textsf{ev-neg}$$

Figure 2.8: Natural semantics for a functional programming language

```
eval : term -> term -> type.

ev_0 : eval 0 0.
ev_s : eval (s M) (s V) <- eval M V.

ev_pair : eval (pair M N) (pair V V') <- eval M V <- eval N V'.
ev_spread : eval (spread M N) V
            <- eval M (pair V1 V2)
            <- eval (N V1 V2) V.
ev_fst : eval (fst M) V1 <- eval M (pair V1 V2).
ev_snd : eval (snd M) V2 <- eval M (pair V1 V2).

ev_inl : eval (inl M) (inl V) <- eval M V.
ev_inr : eval (inr M) (inr V) <- eval M V.
ev_dec_l : eval (decide M Nl Nr) V
            <- eval M (inl V')
            <- eval (Nl V') V.
ev_dec_r : eval (decide M Nl Nr) V
            <- eval M (inr V')
            <- eval (Nr V') V.

ev_lam : eval (lam M) (lam M).
ev_app_lam : eval (app M N) V
             <- eval M (lam M')
             <- eval N V1
             <- eval (M' V1) V.
ev_pr : eval (nat_ind Mz Ms) (nat_ind Mz Ms).
ev_pr_z : eval (app M N) V
          <- eval M (nat_ind Mz Ms)
          <- eval N z
          <- eval Mz V.
ev_pr_s : eval (app M N) V
          <- eval M (nat_ind Mz Ms)
          <- eval N (s N')
          <- eval (app (nat_ind Mz Ms) N') M'
          <- eval (Ms N' M') V.

ev_unity : eval unity unity.
ev_axiom : eval axiom axiom.
ev_neg : eval neg neg.
```

Figure 2.9: Call-by-value natural semantics in Elf

Type inference system

In our language types are external to the language of programs, and the type of a program expression is determined by an inference system, or type assignment system. In order to distinguish this object-level notion of type from LF typing, we use the notation $e \in \tau$ to express the typing judgment defined by this inference system. A notion of a context that assigns types to free variables of an expression is needed.

$$\textit{Contexts} \quad \Gamma \quad ::= \quad \cdot \;\mid\; \Gamma, x \in \tau$$

We adopt the convention that a variable may be declared at most once in any context; thus $\Gamma(x)$ may be written for the unique type assigned to x by the context Γ. The typing judgment is $\Gamma \vdash e \in \tau$ (read: in context Γ the expression e has type τ).

Figure 2.10 gives a deductive system defining this judgment.

The implementation of this system in Elf (Figure 2.11) depends on the use of assumptions to represent a context. Instead of declaring a type `context` and a judgment `of : context -> term -> tp -> type`, we declare the judgment `of : term -> tp -> type`. To model the addition of a type assignment to the current context, the Elf program makes a typing assumption. For example, the typing rule for λ-abstraction is:

$$\frac{\Gamma, x \in \tau_1 \vdash e \in \tau_2}{\Gamma \vdash \mathbf{lam}\, x \,.\, e \in \tau_1 \Rightarrow \tau_2}\,\textsf{tp-lam}$$

The corresponding Elf rule is:

```
tp_lam : of (lam M) (arrow A B)
            <- {x:term} of x A -> of (M x) B.
```

The operational semantics of the Elf interpreter models the required treatment of contexts: upon encountering the subgoal

```
{x:term} of x A -> of (M x) B
```

the interpreter creates a new parameter `x` (modelling the convention that a variable may occur at most once in a context) and adds the judgment `of x A` to its stock of rules. It then searches for a deduction of a type assignment for (the normal form of) the term `(M x)`. When the search reaches a subgoal `of x C` for some type `C`, the assumption `of x A` is used to supply a type for the parameter `x`. This models the type assignment rule for variables:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \in \tau}\,\textsf{tp-var}$$

The type assignment system supports a form of polymorphism sufficient for expressing ML-style let-polymorphism, although we do not exploit this capability. In the implementation a polymorphic type appears as an LF object of type `tp` containing Elf logic variables, which are subject to instantiation. An example is the identity function $\mathbf{lam}\, x \,.\, x$, which is assigned the type $\tau \Rightarrow \tau$ for

$$\frac{}{\Gamma \vdash 0 \in \mathsf{nat}}\ \textsf{tp-0} \qquad \frac{\Gamma \vdash e \in \mathsf{nat}}{\Gamma \vdash \mathsf{s}(e) \in \mathsf{nat}}\ \textsf{tp-s}$$

$$\frac{\Gamma \vdash e_1 \in \tau_1 \qquad \Gamma \vdash e_2 \in \tau_2}{\Gamma \vdash \langle\, e_1, e_2 \,\rangle \in \tau_1 \times \tau_2}\ \textsf{tp-pr} \qquad \frac{\Gamma \vdash e \in \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{fst}(e) \in \tau_1}\ \textsf{tp-fst}$$

$$\frac{\Gamma \vdash e \in \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{snd}(e) \in \tau_2}\ \textsf{tp-snd} \qquad \frac{\Gamma \vdash e_1 \in \tau_1 \times \tau_2 \qquad \Gamma, x \in \tau_1, y \in \tau_2 \vdash e_2 \in \tau}{\Gamma \vdash \mathbf{spread}(e_1;\ x, y\,.\,e_2) \in \tau}\ \textsf{tp-spread}$$

$$\frac{\Gamma \vdash e \in \tau_1}{\Gamma \vdash \mathbf{inl}(e) \in \tau_1 \mid \tau_2}\ \textsf{tp-inl} \qquad \frac{\Gamma \vdash e \in \tau_2}{\Gamma \vdash \mathbf{inr}(e) \in \tau_1 \mid \tau_2}\ \textsf{tp-inr}$$

$$\frac{\Gamma \vdash e_1 \in \tau_1 \mid \tau_2 \qquad \Gamma, x \in \tau_1 \vdash e_2 \in \tau \qquad \Gamma, x \in \tau_2 \vdash e_3 \in \tau}{\Gamma \vdash \mathbf{decide}(e_1;\ x\,.\,e_2;\ x\,.\,e_3) \in \tau}\ \textsf{tp-dec}$$

$$\frac{\Gamma, x \in \tau_1 \vdash e \in \tau_2}{\Gamma \vdash \mathbf{lam}\, x\,.\,e \in \tau_1 \Rightarrow \tau_2}\ \textsf{tp-lam} \qquad \frac{\Gamma \vdash e_1 \in \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash e_2 \in \tau_1}{\Gamma \vdash \mathbf{app}(e_1, e_2) \in \tau_2}\ \textsf{tp-app}$$

$$\frac{\Gamma \vdash e_1 \in \tau \qquad \Gamma, x \in \mathsf{nat}, y \in \tau \vdash e_2 \in \tau}{\Gamma \vdash \mathbf{nat\_ind}(e_1;\ x, y\,.\,e_2) \in \mathsf{nat} \Rightarrow \tau}\ \textsf{tp-prec} \qquad \frac{}{\Gamma \vdash () \in \mathsf{unit}}\ \textsf{tp-unit}$$

$$\frac{\Gamma \vdash e \in \mathsf{void}}{\Gamma \vdash \mathbf{any}(e) \in \tau}\ \textsf{tp-any} \qquad \frac{}{\Gamma \vdash \mathbf{axiom} \in \mathsf{atom}}\ \textsf{tp-axiom}$$

$$\frac{}{\Gamma \vdash \mathbf{neg} \in \tau \Rightarrow \mathsf{void}}\ \textsf{tp-neg} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x \in \tau}\ \textsf{tp-var}$$

Figure 2.10: Type assignment system for the core language

```
of : term -> tp -> type.

tp_0 : of 0 nat.
tp_s : of (s M) nat <- of M nat.

tp_pair : of (pair M N) (* A B) <- of M A <- of N B.
tp_fst :  of (fst M) A <- of M (* A B).
tp_snd :  of (snd M) B <- of M (* A B).
tp_spread : of (spread Mpr N) C
            <- of Mpr (* A B)
            <- {x} of x A -> {y} of y B -> of (N x y) C.

tp_inl :  {B} of (inl M) (| A B) <- of M A.
tp_inr :  {A} of (inr M) (| A B) <- of M B.
tp_dec : of (decide M Nl Nr) C
         <- of M (| A B)
         <- ({x:term} of x A -> of (Nl x) C)
         <- ({x:term} of x B -> of (Nr x) C).

tp_lam : of (lam M) (arrow A B)
            <- {x:term} of x A -> of (M x) B.
tp_app : of (app M N) B <- of M (arrow A B) <- of N A.

tp_prec : of (nat_ind Mz Ms) (arrow nat A)
          <- of Mz A
          <- ({x} of x nat -> {y} of y A -> of (Ms x y) A).

tp_unity : of unity unit.
tp_axiom : of axiom atom.

tp_any : {A} of (any M) A <- of M void.
tp_neg :  {A} of neg (arrow A void).
```

Figure 2.11: Elf implementation of type assignment

$$\begin{tikzcd}
\text{Proof } \mathcal{P} \arrow[r, "\vdash \mathcal{P} : \mathsf{pf}\,\Phi"] \arrow[d, "\vdash u : \mathcal{P} \Downarrow M"'] & \text{Proposition } \Phi \arrow[d, "\vdash w : \Phi \Downarrow^t \tau"] \\
\text{Program } M \arrow[r, "\vdash v : M \in \tau"'] & \text{Type } \tau
\end{tikzcd}$$

Figure 2.12: Elf implementation of proofs-as-programs

any type τ. When presented with the query `of (lam [x] x) T`, Elf succeeds with a substitution of the form `T = (arrow A A)` – `A` is an internally created logic variable; the substitution constrains any ground term found for `T` during further search to be a representation of an object type $\tau \Rightarrow \tau$ for some τ.

2.3.2 Extraction

In this section we define judgments $\Downarrow$ and $\Downarrow^t$ for program and type extraction; this completes an implementation of the proofs-as-programs view of constructive logic, as shown in Figure 2.12. At each vertex of the diagram is an LF object encoding a proof, proposition, program, or (object programming language) type. These objects are related by LF typing assertions – the turnstile $\vdash$ here means derivability in LF. The minuscules u, v, and w should be read existentially: they are the terms found by the Elf interpreter as it searches for proofs of the judgments $\Downarrow$, $\Downarrow^t$, and $\in$. We have already described the judgments pf (proof well-formedness) and $\in$ (object language type assignment). The judgments $\mathcal{P} \Downarrow M$ (read: the program M is extracted from the proof $\mathcal{P}$) and $\Phi \Downarrow^t \tau$ (read: the type τ is extracted from the proposition Φ) are defined so that whenever $\vdash \mathcal{P} : \mathsf{pf}\,\Phi$ and $\vdash u : \mathcal{P} \Downarrow M$ and $\vdash w : \Phi \Downarrow^t \tau$ then $\vdash v : M \in \tau$. (However, there may be other types that can be inferred for M; for example, for **lam** $x \,.\, x$ we can infer $M \in \tau \Rightarrow \tau$ for any type τ.) Moreover if $\vdash u : \mathcal{P} \Downarrow M$ and $\vdash x : M \hookrightarrow V$ then there is a proof $\mathcal{P}'$ such that $\vdash \mathcal{P}' : \mathsf{pf}\,\Phi$ and $\vdash x' : \mathcal{P}' \Downarrow V$. These results are proved and the proofs partially internalized as Elf programs in Chapter 3.

The program extraction judgment implicitly defines a kind of realizability interpretation, if realizability is taken to interpret sentences of the object logic in the domain of expressions of the programming language. The deductive system for establishing the judgment corresponds in its essentials to axiomatic characterizations of realizability such as that found in Troelstra [TvD88]. However, the core object logic is too weak to internalize the realizability relation as is done in other systems. We leave it implicit and rely on reasoning at the metalevel to establish correctness.

The encodings of proofs and functional programs as LF objects facilitate the treatment of type

$$\frac{}{\top \Downarrow^t \mathsf{unit}}\,\mathsf{xt}\top \qquad \frac{}{\bot \Downarrow^t \mathsf{void}}\,\mathsf{xt}\bot$$

$$\frac{A \Downarrow^t \tau}{\neg A \Downarrow^t \tau \Rightarrow \mathsf{void}}\,\mathsf{xt}\neg \qquad \frac{A \Downarrow^t \tau_1 \qquad B \Downarrow^t \tau_2}{A \wedge B \Downarrow^t \tau_1 \times \tau_2}\,\mathsf{xt}\wedge$$

$$\frac{A \Downarrow^t \tau_1 \qquad B \Downarrow^t \tau_2}{A \vee B \Downarrow^t \tau_1 \mid \tau_2}\,\mathsf{xt}\vee \qquad \frac{A \Downarrow^t \tau_1 \qquad B \Downarrow^t \tau_2}{A \supset B \Downarrow^t \tau_1 \Rightarrow \tau_2}\,\mathsf{xt}\supset$$

$$\frac{A \Downarrow^t \tau}{\forall x \,.\, A \Downarrow^t \mathsf{nat} \Rightarrow \tau}\,\mathsf{xt}\forall \qquad \frac{A \Downarrow^t \tau}{\exists x \,.\, A \Downarrow^t \mathsf{nat} \times \tau}\,\mathsf{xt}\exists$$

$$\frac{}{t_1 = t_2 \Downarrow^t \mathsf{atom}}\,\mathsf{xt}{=}$$

Figure 2.13: Type extraction

and program extraction as deductive systems. We first give naive formulations of these systems that extract types that mimic closely the structure of propositions and programs that mimic closely the structure of proofs. For the pure first-order logic fragment of our object logic (without arithmetic) the rules of the type extraction system are precisely congruent to the syntax of propositions; the rules of the program extraction system are congruent to the inference rules of the logic. Then we discuss the distinction between proofs with computational content and those without it, and present a modification of extraction that removes computationally useless subterms of the extracted program.

Naive extraction

The judgment $\Downarrow^t$ relates propositions of FOL and types of the programming language as explained in section 2.3.1. It is defined by the rules of Figure 2.13; its translation into Elf is straightforward and does not involve any new principles.

The judgment $\Downarrow$ relates natural deduction FOL proofs and expressions of the programming language. An auxiliary judgment $\Downarrow^i$ establishes the corresponding relation between individual terms of the logic and programming-language expressions (in our simple system, these must be terms representing natural numbers).

The rules for $\Downarrow$ work by a straightforward case analysis of the last inference in the proof under consideration; therefore we present them in terms of proof schemas. A schema of the form

$$\boxed{\begin{array}{c}\mathcal{P}\\ A\end{array}}$$

matches an object proof ending in a formula A; any variables free in A may also occur free in $\mathcal{P}$.

Since $\Downarrow$ is defined inductively over the structure of proof objects, which contain discharged assumptions and bound individual variables, a context of extraction assignments for assumptions and individuals is needed. Variables for individuals are part of the logical language; they are bound by the quantifiers. Variables for discharged assumptions are part of the proof language and are bound by the inference rules that discharge them ($\supset$I, $\vee$E, $\exists$E, and IND). We represent this context as a pair $\langle\, \Gamma, \Delta \,\rangle$, where Γ is a sequence of assignments $x_i \Downarrow^i x_i'$ for individual variables and Δ a sequence of assignments $\boxed{\begin{array}{c}p_i\\ A_i\end{array}} \Downarrow p_i'$. As usual we stipulate that all variables in a context are distinct. Thus we may treat a context as a pair of partial functions on variables, and write $\Gamma(x) = x'$ when $x \Downarrow^i x'$ is an extraction assignment in Γ, and $\Delta\left(\boxed{\begin{array}{c}p\\ A\end{array}}\right) = p'$ when $\boxed{\begin{array}{c}p\\ A\end{array}} \Downarrow p'$ is an extraction assignment in Δ. We extend this convention in the obvious way and also write $\langle\, \Gamma, \Delta \,\rangle(x) = x'$. We write $\langle\, (\Gamma, x \Downarrow^i x'), \Delta \,\rangle$ for the result of extending $\langle\, \Gamma, \Delta \,\rangle$ with the term extraction assignment $x \Downarrow^i x'$, and

$$\langle\, \Gamma, (\Delta, \boxed{\begin{array}{c}p\\ A\end{array}} \Downarrow p') \,\rangle$$

for the result of extending $\langle\, \Gamma, \Delta \,\rangle$ with the proof extraction assignment $\boxed{\begin{array}{c}p\\ A\end{array}} \Downarrow p'$.

Extraction assignments for assumptions lead to schemas of the form

$$\boxed{\begin{array}{c}p\\ A\\ \mathcal{P}\\ B\end{array}}$$

This matches a proof fragment that depends on an assumption A where p is the (hypothetical) proof of A. The context $\langle\, \Gamma, \Delta \,\rangle$ contains an extraction assignment p' for p.

The full form of the extraction judgment is

$$\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ A\end{array}} \Downarrow e$$

(read: in the context $\langle\, \Gamma, \Delta \,\rangle$, e represents the computational content of the proof $\mathcal{P}$ of A.)

The implementation of extraction in Elf uses the same technique for representing the context as in the implementation of type inference. Instead of representing it explicitly, we introduce parameters and assumptions during the search process. Thus the Elf declaration of the extraction judgment encodes a relation on two objects rather than three, leaving the context implicit:

```
extract : |- A -> term -> type.
extract_tm : i -> term -> type.
```

$$\frac{}{\Gamma \vdash \mathrm{zero} \Downarrow^{i} 0}\,\textsf{ex-zero} \qquad \frac{\Gamma \vdash t \Downarrow^{i} e}{\Gamma \vdash \mathrm{succ}\ t \Downarrow^{i} \mathrm{s}(e)}\,\textsf{ex-succ}$$

Figure 2.14: Program extraction from individual terms

$$\frac{\Gamma(x) = x'}{\Gamma \vdash x \Downarrow^{i} x'}\,\textsf{ex-ivar} \qquad \frac{\Delta\left(\boxed{\begin{matrix} p \\ A \end{matrix}}\right) = p'}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{matrix} p \\ A \end{matrix}} \Downarrow p'}\,\textsf{ex-pvar}$$

Figure 2.15: Extraction for individual and proof variables

The full deductive system for program extraction from individual terms and proofs is presented in Figures 2.14 through 2.19. Extraction from individuals is straightforward in the limited system under consideration here. The inference rules for extraction from proofs imitate the structure of the natural deduction inference rules (compare Figure 2.2). We explain a few characteristic rules in detail and discuss their implementation in Elf.

The simplest case for extraction is a proof in the object logic consisting of an inference rule with no premises (essentially an axiom, though natural deduction does not distinguish between axioms and inference rules). An example is:

$$\frac{}{\top}\,\top\mathrm{I}$$

The corresponding extraction rule has the same structure:

$$\frac{}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\frac{}{\top}\,\top\mathrm{I}} \Downarrow ()}\,\textsf{x}\top$$

Its encoding in Elf is straightforward:

```
ex_truei : extract truei unity.
```

Recall that **truei** has been declared as an object of type **|- true**; it represents the proof of $\top$ in the extraction deduction. Similarly, **unity** is an object of type **term** representing the program

$$\frac{\langle \Gamma,\Delta \rangle \vdash \boxed{\begin{array}{c}\mathcal{P}_1\\ A\end{array}} \Downarrow e_1 \qquad \langle \Gamma,\Delta \rangle \vdash \boxed{\begin{array}{c}\mathcal{P}_2\\ B\end{array}} \Downarrow e_2}{\langle \Gamma,\Delta \rangle \vdash \boxed{\dfrac{\begin{array}{cc}\mathcal{P}_1 & \mathcal{P}_2\\ A & B\end{array}}{A \wedge B}\,{\scriptstyle \wedge\mathrm{I}}} \Downarrow \langle e_1, e_2 \rangle}\ \mathrm{x}\wedge\mathrm{I}$$

$$\frac{\langle \Gamma,\Delta \rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ A \wedge B\end{array}} \Downarrow e}{\langle \Gamma,\Delta \rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ \dfrac{A \wedge B}{A}\,{\scriptstyle \wedge\mathrm{E}_L}\end{array}} \Downarrow \mathbf{fst}(e)}\ \mathrm{x}\wedge\mathrm{E}_L \qquad \frac{\langle \Gamma,\Delta \rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ A \wedge B\end{array}} \Downarrow e}{\langle \Gamma,\Delta \rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ \dfrac{A \wedge B}{A}\,{\scriptstyle \wedge\mathrm{E}_R}\end{array}} \Downarrow \mathbf{snd}(e)}\ \mathrm{x}\wedge\mathrm{E}_R$$

$$\frac{\langle \Gamma,\Delta \rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ A\end{array}} \Downarrow e}{\langle \Gamma,\Delta \rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ \dfrac{A}{A \vee B}\,{\scriptstyle \vee\mathrm{I}_L}\end{array}} \Downarrow \mathbf{inl}(e)}\ \mathrm{x}\vee\mathrm{I}_L \qquad \frac{\langle \Gamma,\Delta \rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ B\end{array}} \Downarrow e}{\langle \Gamma,\Delta \rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ \dfrac{B}{A \vee B}\,{\scriptstyle \vee\mathrm{I}_R}\end{array}} \Downarrow \mathbf{inr}(e)}\ \mathrm{x}\vee\mathrm{I}_R$$

$$\frac{\langle \Gamma,\Delta \rangle \vdash \boxed{\begin{array}{c}\mathcal{P}_1\\ A \vee B\end{array}} \Downarrow e_1 \qquad \langle \Gamma,(\Delta,\boxed{\begin{array}{c}p\\ A\end{array}} \Downarrow p') \rangle \vdash \boxed{\begin{array}{c}p\\ A\\ \mathcal{P}_2\\ C\end{array}} \Downarrow e_2 \qquad \langle \Gamma,(\Delta,\boxed{\begin{array}{c}p\\ B\end{array}} \Downarrow p') \rangle \vdash \boxed{\begin{array}{c}p\\ B\\ \mathcal{P}_3\\ C\end{array}} \Downarrow e_3}{\langle \Gamma,\Delta \rangle \vdash \boxed{\dfrac{\begin{array}{ccc} & \overline{A}^{\,p} & \overline{B}^{\,p}\\ \mathcal{P}_1 & \mathcal{P}_2 & \mathcal{P}_3\\ A \vee B & C & C\end{array}}{C}\,{\scriptstyle \vee\mathrm{E}^p}} \Downarrow \mathbf{decide}(e_1;\ p'.e_2;\ p'.e_3)}\ \mathrm{x}\vee\mathrm{E}$$

Figure 2.16: Program extraction, logical rules, part 1

$$\dfrac{\langle\, \Gamma, (\Delta, \boxed{\begin{array}{c} p \\ A \end{array}} \Downarrow p')\,\rangle \vdash \boxed{\begin{array}{c} p \\ A \\ \mathcal{P} \\ B \end{array}} \Downarrow e}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \overset{\overline{\quad}\, p}{A} \\ \mathcal{P} \\ \dfrac{B}{A \supset B}\,{\supset}\mathrm{I}^{p} \end{array}} \Downarrow \mathbf{lam}\, p' . e}\ \mathrm{x{\supset}I}
\qquad
\dfrac{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P}_1 \\ A \supset B \end{array}} \Downarrow e_1 \qquad \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P}_2 \\ A \end{array}} \Downarrow e_2}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\dfrac{\begin{array}{cc} \mathcal{P}_1 & \mathcal{P}_2 \\ A \supset B & A \end{array}}{B}\,{\supset}\mathrm{E}} \Downarrow \mathbf{app}(e_1, e_2)}\ \mathrm{x{\supset}E}$$

$$\dfrac{\langle\, (\Gamma, x \Downarrow^{i} x'), \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P} \\ A \end{array}} \Downarrow e}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P} \\ \dfrac{A}{\forall x . A}\,\forall\mathrm{I} \end{array}} \Downarrow \mathbf{lam}\, x' . e}\ \mathrm{x\forall I}
\qquad
\dfrac{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P} \\ \forall x . A \end{array}} \Downarrow e_1 \qquad \Gamma \vdash t \Downarrow^{i} e_2}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P} \\ \dfrac{\forall x . A}{[t/x]A}\,\forall\mathrm{E} \end{array}} \Downarrow \mathbf{app}(e_1, e_2)}\ \mathrm{x\forall E}$$

$$\dfrac{\Gamma \vdash t \Downarrow^{i} e_1 \qquad \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P} \\ [t/x]A \end{array}} \Downarrow e_2}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P} \\ \dfrac{[t/x]A}{\exists x . A}\,\exists\mathrm{I} \end{array}} \Downarrow \langle\, e_1, e_2 \,\rangle}\ \mathrm{x\exists I}$$

$$\dfrac{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P} \\ \exists x . A \end{array}} \Downarrow e_1 \qquad \langle\, (\Gamma, x \Downarrow^{i} x'), (\Delta, \boxed{\begin{array}{c} p \\ A \end{array}} \Downarrow p')\,\rangle \vdash \boxed{\begin{array}{c} p \\ A \\ \mathcal{Q} \\ C \end{array}} \Downarrow e_2}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\dfrac{\begin{array}{cc} & \overset{\overline{\quad}\, p}{A} \\ \mathcal{P} & \mathcal{Q} \\ \exists x . A & C \end{array}}{C}\,\exists\mathrm{E}^{p}} \Downarrow \mathbf{spread}(e_1;\ x', p' . e_2)}\ \mathrm{x\exists E}$$

Figure 2.17: Program extraction, logical rules, part 2

$$\dfrac{\langle\, \Gamma, (\Delta, \boxed{\begin{array}{c} p \\ A \end{array}} \Downarrow p')\,\rangle \vdash \boxed{\begin{array}{c} p \\ A \\ \mathcal{P} \\ \bot \end{array}} \Downarrow e}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{l} \overline{\;\;}^{\,p} \\ A \\ \mathcal{P} \\ \bot \\ \overline{\neg A}\;{}^{\neg \mathrm{I}^{p}} \end{array}} \Downarrow \mathbf{lam}\, p' . e}\ \mathsf{x}\neg\mathrm{I}$$

$$\dfrac{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P}_1 \\ \neg A \end{array}} \Downarrow e_1 \qquad \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P}_2 \\ A \end{array}} \Downarrow e_2}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\dfrac{\begin{array}{cc} \mathcal{P}_1 & \mathcal{P}_2 \\ \neg A & A \end{array}}{\bot}\,{}_{\neg \mathrm{E}}} \Downarrow \mathbf{app}(e_1, e_2)}\ \mathsf{x}\neg\mathrm{E}$$

$$\dfrac{}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\dfrac{}{\top}\,{}^{\top \mathrm{I}}} \Downarrow ()}\ \mathsf{x}\top$$

$$\dfrac{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P} \\ \bot \end{array}} \Downarrow e}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{l} P \\ \dfrac{\bot}{C}\,{}_{\bot \mathrm{E}} \end{array}} \Downarrow \mathbf{any}(e)}\ \mathsf{x}\bot$$

Figure 2.18: Program extraction, logical rules, part 3

$$\frac{}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\frac{}{t = t}\,{}^{=\mathrm{R}}} \Downarrow \mathbf{axiom}}\ \mathrm{x{=}R}
\qquad
\frac{}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P} \\ \dfrac{t_2 = t_1}{t_1 = t_2}\,{}_{=\mathrm{Y}} \end{array}} \Downarrow \mathbf{axiom}}\ \mathrm{x{=}Y}$$

$$\frac{}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P}_1 \qquad\quad \mathcal{P}_2 \\ \dfrac{t_1 = t_2 \qquad t_2 = t_3}{t_1 = t_3}\,{}_{=\mathrm{T}} \end{array}} \Downarrow \mathbf{axiom}}\ \mathrm{x{=}T}
\qquad
\frac{}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P} \\ \dfrac{t_1 = t_2}{\mathrm{succ}\ t_1 = \mathrm{succ}\ t_2}\,{}_{=\mathrm{U}} \end{array}} \Downarrow \mathbf{axiom}}\ \mathrm{x{=}U}$$

$$\frac{}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\frac{}{\neg\mathrm{succ}\ t = \mathrm{zero}}\,{}^{\mathrm{AX0}}} \Downarrow \mathbf{neg}}\ \mathrm{xAX0}
\qquad
\frac{}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P} \\ \dfrac{\mathrm{succ}\ t_1 = \mathrm{succ}\ t_2}{t_1 = t_2}\,{}_{\mathrm{AXS}} \end{array}} \Downarrow \mathbf{axiom}}\ \mathrm{xAXS}$$

$$\frac{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P}_1 \\ [\mathrm{zero}/x]A \end{array}} \Downarrow e_1 \qquad \langle\, (\Gamma, x \Downarrow^{i} x'), (\Delta, \boxed{\begin{array}{c} p \\ A \end{array}} \Downarrow p') \,\rangle \vdash \boxed{\begin{array}{c} p \\ A \\ \mathcal{P} \\ [\mathrm{succ}\ x/x]A \end{array}} \Downarrow e_2}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \qquad\qquad\qquad \dfrac{}{A}\,{}^{p} \\ \dfrac{\begin{array}{cc} \mathcal{P}_1 & \mathcal{P} \\ [\mathrm{zero}/x]A & [\mathrm{succ}\ x/x]A \end{array}}{\forall x\,.\,A}\,{}_{\mathrm{IND}^{p}} \end{array}} \Downarrow \mathbf{nat\_ind}(e_1;\ x', p'\,.\,e_2)}\ \mathrm{xIND}$$

Figure 2.19: Program extraction, arithmetic rules

expression (). The implicit propositional argument `true` of the `extract` judgment is reconstructed by Elf during the type checking of this rule from the type of `truei`.

In the case of a proof ending in an inference rule that has premises, but neither discharges an assumption nor binds an individual variable, extraction simply descends through the proof, extracting program expressions for the subproof(s), and applies the appropriate constructor or destructor to the resulting program expression. To express this as an inference rule, we use schematic variables for subproofs – these are distinct from the bound variables associated with assumptions. In the notation

$$\begin{matrix}\mathcal{P}\\ A\end{matrix}$$

$\mathcal{P}$ stands for the proof with conclusion A.

Extraction from the rule of conjunction introduction is an example:

$$\frac{\langle \Gamma,\Delta \rangle \vdash \boxed{\begin{matrix}\mathcal{P}_1\\ A\end{matrix}} \Downarrow e_1 \qquad \langle \Gamma,\Delta \rangle \vdash \boxed{\begin{matrix}\mathcal{P}_2\\ B\end{matrix}} \Downarrow e_2}{\langle \Gamma,\Delta \rangle \vdash \boxed{\dfrac{\begin{matrix}\mathcal{P}_1 & \mathcal{P}_2\\ A & B\end{matrix}}{A \wedge B}\,{\scriptstyle \wedge I}} \Downarrow \langle e_1, e_2 \rangle}\ \text{x}\wedge\text{I}$$

The encoding in Elf is again straightforward, simply introducing two subgoals:

```
ex_andi : extract (andi P1 P2) (pair M1 M2)
            <- extract P1 M1 <- extract P2 M2.
```

Again term reconstruction permits the omission of the implicit propositional argument, since `andi` has been declared with type `|- A -> |- B -> |- (and A B)`.

The $\supset$I rule is a simple example of the treatment of the discharge of assumptions by adding an extraction assignment to the context $\langle \Gamma, \Delta \rangle$.

$$\frac{\langle \Gamma,(\Delta, \boxed{\begin{matrix}p\\ A\end{matrix}} \Downarrow p') \rangle \vdash \boxed{\begin{matrix}p\\ A\\ \mathcal{P}\\ B\end{matrix}} \Downarrow e}{\langle \Gamma,\Delta \rangle \vdash \boxed{\dfrac{\begin{matrix}\overline{A}^{\,p}\\ \mathcal{P}\\ B\end{matrix}}{A \supset B}\,{\scriptstyle \supset I^p}} \Downarrow \mathbf{lam}\, p' \,.\, e}\ \text{x}\supset\text{I}$$

The assignment

$$\boxed{\begin{matrix}p\\ A\end{matrix}} \Downarrow p'$$

associates the hypothetical proof p of the assumption A with the program variable p' bound by the **lam**-abstraction. As a result, when a goal of extracting an expression from p occurs in the deduction, the rule **ex-pvar** for proof variables applies.

The Elf implementation models the context $\langle\, \Gamma, \Delta \,\rangle$ at the meta-level by introducing assumptions. The Elf encoding of the rule x⊃I is:

```
ex_impliesi : extract (impliesi P) (lam M)
               <- {p:|- A} {p':term} extract p p' -> extract (P p) (M p').
```

In contrast to the assumptions introduced in the implementation of the type assignment system, which quantify over one parameter representing a bound program variable, here we quantify over two parameters: `p` represents a bound proof variable and `p'` represents a bound program variable.

The treatment of bound individual variables in proofs is much the same, but the extraction assignment added to the context is modelled as the `extract-tm` judgment. An example is the extraction rule for ∀I:

$$\dfrac{\langle\, (\Gamma, x \Downarrow^{i} x'), \Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ A\end{array}} \Downarrow e}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ \dfrac{A}{\forall x \,.\, A}\,\forall\mathrm{I}\end{array}} \Downarrow \mathbf{lam}\, x' \,.\, e}\;\mathrm{x}\forall\mathrm{I}$$

The Elf implementation is:

```
ex_foralli : extract (foralli P) (lam M)
               <- {x:i} {x':term} (extract_tm x x' -> extract (P x) (M x')).
```

Program simplification during extraction

As noted above, naive extraction results in a program containing many subterms that carry no computationally useful information. Intuitively speaking, this is because the only logical formulas whose proofs involve choice are disjunctions and existential quantifications. A proof of $A \vee B$ provides a method for deciding whether A or B holds; a proof of $\exists x \,.\, A$ provides a method for finding a particular natural number t (the "witness") such that $[t/x]A$ holds. The inference rules for the other logical connectives merely combine the proofs of their components in an appropriate way. If the components are void of computational content, so is the resulting proof. We call formulas free of $\exists$ and $\vee$ *uninformative*. By extension we call their proofs, the object types extracted from them, and the programs extracted from the proofs uninformative as well.

The work described here is an adaptation to the Elf setting of the basic ideas of *modified realizability* developed for the Calculus of Constructions by Paulin-Mohring [PM89] which in turn takes from the PX system [Hay90] the idea of syntactically defining a class of content-free terms. Sasaki [Sas86] develops these ideas for Nuprl. The negative formulas of Schwichtenberg [Sch82], [Sch85] are used in a similar way to decrease the complexity of realizing terms.

We define extraction procedures for types and programs that simplify the extracted terms to remove uninformative subterms while retaining computationally useful information. As for naive extraction, the extracted programs are well-typed and the extracted types can be inferred for them by the type inference system of Figure 2.10.

In order to preserve well-typedness, we distinguish between positive and negative uninformative formulas.

The positive uninformative formulas are the *Harrop*[Har60] formulas:

$$\textit{Harrop formulas} \quad H \quad ::= \quad \top \ \mid \ t_1 = t_2 \ \mid \ H \wedge H \ \mid \ \forall x \,.\, H \ \mid \ A \supset H$$

where A is any formula.

For these formulas we extract the unit element (). Programs can be further simplified by noting that an expression extracted from $A \supset B$, where A is Harrop, will have type $\textsf{unit} \Rightarrow \tau$, where τ is the type extracted from B. But this type carries no more information than τ itself, so we may as well extract a term of type τ instead. Similar considerations apply to extraction from $A \wedge B$ where either A or B is Harrop.

As a small example, consider a program to compute the predecessor, extracted from a proof of $\forall x \,.\, x = \text{zero} \vee (\exists y \,.\, x = \text{succ}\ (y))$. The equalities are Harrop formulas; their proofs are represented in the program by the special constant **axiom**.

$$\textbf{nat\_ind}(\,(\textbf{inl}(\textbf{axiom}))\,;\ x, v \,.\, \textbf{inr}(\langle\, x, \textbf{axiom}\,\rangle))$$

The type of this expression is $\textsf{nat} \Rightarrow (\textsf{atom} \mid (\textsf{nat} \times \textsf{atom}))$. The occurrences of the type $\textsf{atom}$ represent the uninformative parts of the proof.

When the proof is simplified by extracting unit terms () for Harrop formulas, the resulting program has type $\textsf{nat} \Rightarrow (\textsf{unit} \mid \textsf{nat})$. The simplification removes the purely logical part of the proof, leaving the left and right injections and the witness, which is the predecessor value we want to compute.

$$\textbf{nat\_ind}(\,\textbf{inl}(())\,;\ x, v \,.\, \textbf{inr}(x))$$

The negative (uninformative) formulas are

$$\textit{Negative formulas} \quad N \quad ::= \quad \bot \ \mid \ \neg A$$

where A is any formula.

These are treated separately from the Harrop formulas because of the intuitionistic absurdity rule $\bot$E, which allows the deduction of any formula from a proof of $\bot$. Recall the corresponding typing rule:

$$\frac{\Gamma \vdash e \in \textsf{void}}{\Gamma \vdash \textbf{any}(e) \in \tau}\ \textsf{tp-any}$$

Terms extracted from proofs of negations have type $\tau \Rightarrow \textsf{void}$; we cannot simply reduce these to the unit element () if they appear in **any**(e), or the result cannot be typed. Nor can we extract an element of type $\textsf{void}$ if the semantics of the language is to faithfully reflect a consistent logic.

Consider another predecessor program, this time extracted (naively) from an inductive proof of $\forall x \,.\, \neg x = \text{zero} \supset \exists y \,.\, x = \text{succ}\ y$. The absurdity rule $\bot$E occurs in the base case of the induction.

$$\frac{}{\mathrm{Harrop}\,\top}\ \textsf{h-true} \qquad \frac{}{\mathrm{Harrop}\,t_1 = t_2}\ \textsf{h-eq}$$

$$\frac{\mathrm{Harrop}\,A \qquad \mathrm{Harrop}\,B}{\mathrm{Harrop}\,A \wedge B}\ \textsf{h-and} \qquad \frac{\mathrm{Harrop}\,A}{\mathrm{Harrop}\,\forall x\,.\,A}\ \textsf{h-forall}$$

$$\frac{\mathrm{Harrop}\,B}{\mathrm{Harrop}\,A \supset B}\ \textsf{h-implies}$$

Figure 2.20: Harrop formulas

$$\mathbf{nat\_ind}(\mathbf{lam}\,v\,.\,\mathbf{any}(\mathbf{app}(v, \mathbf{axiom}));\ x, p\,.\,\mathbf{lam}\,v\,.\,\langle\, x, \mathbf{axiom}\,\rangle)$$

The extracted type is $\mathsf{nat} \Rightarrow (\mathsf{atom} \Rightarrow \mathsf{void}) \Rightarrow (\mathsf{nat} \times \mathsf{atom})$. The program subexpression of interest is $\mathbf{lam}\,v\,.\,\mathbf{any}(\mathbf{app}(v, \mathbf{axiom}))$, extracted from the base case of the induction. The component $\mathsf{atom} \Rightarrow \mathsf{void}$ of the type corresponds to the bound variable v, which (for the base case) represents a proof of $\neg\mathrm{zero} = \mathrm{zero}$, which is not provable in a consistent system. Thus this case returns a function which it would be an error to apply; that is why the function body is $\mathbf{any}(\mathbf{app}(v, \mathbf{axiom}))$, which cannot be evaluated. Any term that could be bound to v would have no computational content, so it is desirable to eliminate it altogether. But we must still insure that an error is signalled in the base case, which would not happen if we simplified the subexpression to $()$; and $\mathbf{any}(())$ is not typable. Instead we extract the constant $\mathbf{neg}$ for any proof of a negative formula, and $\mathbf{app}(\mathbf{neg}, ())$ for any proof of $\bot$, so the simplified subexpression is $\mathbf{any}(\mathbf{app}(\mathbf{neg}, ()))$, which is typable. The whole simplified program is:

$$\mathbf{nat\_ind}(\mathbf{any}(\mathbf{app}(\mathbf{neg}, ()));\ x, p\,.\,x)$$

with type $\mathsf{nat} \Rightarrow \mathsf{nat}$.

As before, we give the implementation of simplification during extraction of types and programs in the form of deductive systems. We define the following judgments:

1. Harrop A (A is a Harrop formula): Figure 2.20
2. Uninf A (A is an uninformative formula): Figure 2.21
3. Inf A (A is an informative formula): Figure 2.22
4. $A \Downarrow^t_s \tau$ (type extraction/simplification): Figure 2.23

$$\frac{\text{Harrop}\,A}{\text{Uninf}\,A}\,\textsf{unin-h} \qquad \frac{}{\text{Uninf}\,\neg A}\,\textsf{unin-n}$$

$$\frac{}{\text{Uninf}\,\bot}\,\textsf{unin-f} \qquad \frac{\text{Uninf}\,A \qquad \text{Uninf}\,B}{\text{Uninf}\,A \wedge B}\,\textsf{unin-and}$$

$$\frac{\text{Uninf}\,A}{\text{Uninf}\,\forall x\,.\,A}\,\textsf{unin-forall} \qquad \frac{\text{Uninf}\,B}{\text{Uninf}\,A \supset B}\,\textsf{unin-implies}$$

Figure 2.21: Uninformative formulas

$$\frac{}{\text{Inf}\,A \vee B}\,\textsf{inf-or} \qquad \frac{}{\text{Inf}\,\exists x\,.\,A}\,\textsf{inf-exists}$$

$$\frac{\text{Inf}\,A}{\text{Inf}\,A \wedge B}\,\textsf{inf-andl} \qquad \frac{\text{Inf}\,B}{\text{Inf}\,A \wedge B}\,\textsf{inf-andr}$$

$$\frac{\text{Inf}\,A}{\text{Inf}\,\forall x\,.\,A}\,\textsf{inf-forall} \qquad \frac{\text{Inf}\,B}{\text{Inf}\,A \supset B}\,\textsf{inf-implies}$$

Figure 2.22: Informative formulas

$$\frac{\mathrm{Harrop}\,A}{A \Downarrow^t_s \mathsf{unit}}\,\mathsf{xst\text{-}H} \qquad\qquad \frac{}{\bot \Downarrow^t_s \mathsf{void}}\,\mathsf{xst}\bot$$

$$\frac{A \Downarrow^t_s \tau}{\neg A \Downarrow^t_s \tau \Rightarrow \mathsf{void}}\,\mathsf{xst}\neg \qquad\qquad \frac{\mathrm{Inf}\,A \quad A \Downarrow^t_s \tau_1 \qquad \mathrm{Inf}\,B \quad B \Downarrow^t_s \tau_2}{A \wedge B \Downarrow^t_s \tau_1 \times \tau_2}\,\mathsf{xst}\wedge$$

$$\frac{\mathrm{Inf}\,A \quad A \Downarrow^t_s \tau \qquad \mathrm{Uninf}\,B}{A \wedge B \Downarrow^t_s \tau}\,\mathsf{xst}\wedge\mathrm{L} \qquad\qquad \frac{\mathrm{Uninf}\,A \qquad \mathrm{Inf}\,B \quad B \Downarrow^t_s \tau}{A \wedge B \Downarrow^t_s \tau}\,\mathsf{xst}\wedge\mathrm{R}$$

$$\frac{A \Downarrow^t_s \tau_1 \qquad B \Downarrow^t_s \tau_2}{A \vee B \Downarrow^t_s \tau_1 \mid \tau_2}\,\mathsf{xst}\vee \qquad\qquad \frac{\mathrm{Inf}\,A \quad A \Downarrow^t_s \tau_1 \qquad \mathrm{Inf}\,B \quad B \Downarrow^t_s \tau_2}{A \supset B \Downarrow^t_s \tau_1 \Rightarrow \tau_2}\,\mathsf{xst}\supset$$

$$\frac{\mathrm{Uninf}\,A \qquad \mathrm{Inf}\,B \quad B \Downarrow^t_s \tau}{A \supset B \Downarrow^t_s \tau}\,\mathsf{xst}\supset\mathrm{R} \qquad\qquad \frac{\mathrm{Inf}\,A \quad A \Downarrow^t_s \tau}{\forall x\,.\,A \Downarrow^t_s \mathsf{nat} \Rightarrow \tau}\,\mathsf{xst}\forall$$

$$\frac{\mathrm{Inf}\,A \qquad A \Downarrow^t_s \tau}{\exists x\,.\,A \Downarrow^t_s \mathsf{nat} \times \tau}\,\mathsf{xst}\exists \qquad\qquad \frac{\mathrm{Uninf}\,A}{\exists x\,.\,A \Downarrow^t_s \mathsf{nat}}\,\mathsf{xst}\exists\mathrm{L}$$

Figure 2.23: Type extraction/simplification

$$\frac{\text{Uninf } A}{\langle \Gamma, \Delta \rangle \vdash \boxed{\begin{matrix}\mathcal{P}\\ A\end{matrix}} \Downarrow_s ()}\ \mathbf{xsH} \qquad \frac{}{\langle \Gamma, \Delta \rangle \vdash \boxed{\begin{matrix}\mathcal{P}\\ \neg A\end{matrix}} \Downarrow_s \mathbf{neg}}\ \mathbf{xs}\neg$$

$$\frac{}{\langle \Gamma, \Delta \rangle \vdash \boxed{\begin{matrix}\mathcal{P}\\ \bot\end{matrix}} \Downarrow_s \mathbf{app}(\mathbf{neg}, ())}\ \mathbf{xs}\bot$$

Figure 2.24: Program extraction/simplification for uninformative proofs

5. $A \Downarrow_s e$ (program extraction/simplification): Figures 2.24, 2.25, 2.26, and 2.27

The first three are simple syntactic properties of formulas: (1) is a straightforward deductive formulation of the grammar of Harrop formulas given above; (2) defines uninformative formulas as formulas built up from Harrop and negative formulas; and (3) is the complement of (2). The main judgments are the extraction/simplification of types and programs, guided by the syntactic analysis given by the three auxiliary judgments.

The deductive systems for both type extraction and program extraction are modifications to the naive extraction.

In type extraction most of the rules of Figure 2.13 are retained, but premises are added to them to ensure that all subformulas are informative. When the whole formula is uninformative no analysis of subformulas is done; we merely need to distinguish between Harrop and negative uninformative formulas. For conjunction, implication, and existential quantification extra rules are added to account for uninformative subformulas. No special rule for type extraction from equalities is needed since they are Harrop.

Program extraction is defined similarly although complications arise because of the elimination rules of the logic. Extraction for individual terms and for individual and proof variables is defined in the same way as for naive extraction (Figures 2.14 and 2.15 respectively), so we omit their explicit definitions. For uninformative formulas and the introduction rules of the logic, program extraction has the same form as type extraction. For logical disjunction, extraction is defined as for the naive case. Even if both A and B are uninformative, a proof of $A \vee B$ always contains at least the computation of which case holds, so the extracted program expression must be a left or right injection into a disjoint union type, although that type may be simpler than in the case of naive extraction.

$$\frac{\text{Inf}\,A \quad \langle \Gamma,\Delta\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}_1\\ A\end{array}} \Downarrow_s e_1 \quad \text{Inf}\,B \quad \langle \Gamma,\Delta\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}_2\\ B\end{array}} \Downarrow_s e_2}{\langle \Gamma,\Delta\rangle \vdash \boxed{\dfrac{\begin{array}{cc}\mathcal{P}_1 & \mathcal{P}_2\\ A & B\end{array}}{A \wedge B}{\scriptstyle \wedge\text{I}}} \Downarrow_s \langle e_1, e_2\rangle}\;\text{xs}\wedge\text{I}$$

$$\frac{\text{Inf}\,A \quad \langle \Gamma,\Delta\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}_1\\ A\end{array}} \Downarrow_s e \quad \text{Uninf}\,B}{\langle \Gamma,\Delta\rangle \vdash \boxed{\dfrac{\begin{array}{cc}\mathcal{P}_1 & \mathcal{P}_2\\ A & B\end{array}}{A \wedge B}{\scriptstyle \wedge\text{I}}} \Downarrow_s e}\;\text{xs}\wedge\text{IL}$$

$$\frac{\text{Uninf}\,A \quad \text{Inf}\,B \quad \langle \Gamma,\Delta\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}_2\\ B\end{array}} \Downarrow_s e}{\langle \Gamma,\Delta\rangle \vdash \boxed{\dfrac{\begin{array}{cc}\mathcal{P}_1 & \mathcal{P}_2\\ A & B\end{array}}{A \wedge B}{\scriptstyle \wedge\text{I}}} \Downarrow_s e}\;\text{xs}\wedge\text{IR}$$

$$\frac{\text{Inf}\,A \quad \text{Inf}\,B \quad \langle \Gamma,\Delta\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ A \wedge B\end{array}} \Downarrow_s e}{\langle \Gamma,\Delta\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ \dfrac{A \wedge B}{A}{\scriptstyle \wedge\text{E}_L}\end{array}} \Downarrow_s \mathbf{fst}(e)}\;\text{xs}\wedge\text{E}_L$$

$$\frac{\text{Inf}\,A \quad \text{Uninf}\,B \quad \langle \Gamma,\Delta\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ A \wedge B\end{array}} \Downarrow_s e}{\langle \Gamma,\Delta\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ \dfrac{A \wedge B}{A}{\scriptstyle \wedge\text{E}_L}\end{array}} \Downarrow_s e}\;\text{xs}\wedge\text{E}_L\text{L}$$

$$\frac{\text{Inf}\,A \quad \text{Inf}\,B \quad \langle \Gamma,\Delta\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ A \wedge B\end{array}} \Downarrow_s e}{\langle \Gamma,\Delta\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ \dfrac{A \wedge B}{A}{\scriptstyle \wedge\text{E}_R}\end{array}} \Downarrow_s \mathbf{snd}(e)}\;\text{xs}\wedge\text{E}_R$$

$$\frac{\text{Uninf}\,A \quad \text{Inf}\,B \quad \langle \Gamma,\Delta\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ A \wedge B\end{array}} \Downarrow_s e}{\langle \Gamma,\Delta\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ \dfrac{A \wedge B}{A}{\scriptstyle \wedge\text{E}_R}\end{array}} \Downarrow_s e}\;\text{xs}\wedge\text{E}_R\text{R}$$

Figure 2.25: Program extraction/simplification, conjunction rules

$$\frac{\langle \Gamma, \Delta \rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ A\end{array}} \Downarrow_s e}{\langle \Gamma, \Delta \rangle \vdash \boxed{\dfrac{\begin{array}{c}\mathcal{P}\\ A\end{array}}{A \vee B}\,{\scriptstyle \vee\mathrm{I}_L}} \Downarrow_s \mathbf{inl}(e)}\ \mathsf{xs}\vee\mathrm{I}_L \qquad \frac{\langle \Gamma, \Delta \rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ B\end{array}} \Downarrow_s e}{\langle \Gamma, \Delta \rangle \vdash \boxed{\dfrac{\begin{array}{c}\mathcal{P}\\ B\end{array}}{A \vee B}\,{\scriptstyle \vee\mathrm{I}_R}} \Downarrow_s \mathbf{inr}(e)}\ \mathsf{xs}\vee\mathrm{I}_R$$

$$\frac{\langle \Gamma, \Delta \rangle \vdash \boxed{\begin{array}{c}\mathcal{P}_1\\ A \vee B\end{array}} \Downarrow_s e_1 \qquad \langle \Gamma, (\Delta, \boxed{\begin{array}{c}p\\ A\end{array}} \Downarrow_s p')\rangle \vdash \boxed{\begin{array}{c}p\\ A\\ \mathcal{P}_2\\ C\end{array}} \Downarrow_s e_2 \qquad \langle \Gamma, (\Delta, \boxed{\begin{array}{c}p\\ B\end{array}} \Downarrow_s p')\rangle \vdash \boxed{\begin{array}{c}p\\ B\\ \mathcal{P}_3\\ C\end{array}} \Downarrow_s e_3}{\langle \Gamma, \Delta \rangle \vdash \boxed{\dfrac{\begin{array}{c}\\ \mathcal{P}_1\\ A \vee B\end{array} \quad \begin{array}{c}\overset{—p}{A}\\ \mathcal{P}_2\\ C\end{array} \quad \begin{array}{c}\overset{—p}{B}\\ \mathcal{P}_3\\ C\end{array}}{C}\,{\scriptstyle \vee\mathrm{E}^p}} \Downarrow_s \mathbf{decide}(e_1;\ p'.e_2;\ p'.e_3)}\ \mathsf{xs}\vee\mathrm{E}$$

Figure 2.26: Program extraction/simplification, disjunction rules

$$\dfrac{\mathrm{Inf}\,A \quad \mathrm{Inf}\,B \quad \langle\, \Gamma, (\Delta, \boxed{\begin{array}{c} p \\ A \end{array}} \Downarrow_s p') \,\rangle \vdash \boxed{\begin{array}{c} p \\ A \\ \mathcal{P} \\ B \end{array}} \Downarrow_s e}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} -p \\ A \\ \mathcal{P} \\ \dfrac{B}{A \supset B}\,{\supset}\mathrm{I}^p \end{array}} \Downarrow_s \mathbf{lam}\, p' .\, e}\;\mathrm{xs}{\supset}\mathrm{I}$$

$$\dfrac{\mathrm{Uninf}\,A \quad \mathrm{Inf}\,B \quad \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} p \\ A \\ \mathcal{P} \\ B \end{array}} \Downarrow_s e}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} -p \\ A \\ \mathcal{P} \\ \dfrac{B}{A \supset B}\,{\supset}\mathrm{I}^p \end{array}} \Downarrow_s e}\;\mathrm{xs}{\supset}\mathrm{IR}$$

$$\dfrac{\mathrm{Inf}\,A \quad \mathrm{Inf}\,B \quad \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P}_1 \\ A \supset B \end{array}} \Downarrow_s e_1 \quad \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P}_2 \\ A \end{array}} \Downarrow_s e_2}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\dfrac{\begin{array}{cc} \mathcal{P}_1 & \mathcal{P}_2 \\ A \supset B & A \end{array}}{B}\,{\supset}\mathrm{E}} \Downarrow_s \mathbf{app}(e_1, e_2)}\;\mathrm{xs}{\supset}\mathrm{E}$$

$$\dfrac{\mathrm{Uninf}\,A \quad \mathrm{Inf}\,B \quad \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P}_1 \\ A \supset B \end{array}} \Downarrow_s e}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\dfrac{\begin{array}{cc} \mathcal{P}_1 & \mathcal{P}_2 \\ A \supset B & A \end{array}}{B}\,{\supset}\mathrm{E}} \Downarrow_s e}\;\mathrm{xs}{\supset}\mathrm{ER}$$

$$\dfrac{\mathrm{Inf}\,A \quad \langle\, ((\Gamma, x \Downarrow^i x'), \Delta \,\rangle \vdash \boxed{\mathcal{P}} \Downarrow_s e}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P} \\ \dfrac{A}{\forall x \,.\, A}\,\forall\mathrm{I} \end{array}} \Downarrow_s \mathbf{lam}\, x' .\, e}\;\mathrm{x}\forall\mathrm{I}$$

$$\dfrac{\mathrm{Inf}\,A \quad \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P} \\ \forall x \,.\, A \end{array}} \Downarrow_s e_1 \qquad \Gamma \vdash t \Downarrow^i}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P} \\ \dfrac{\forall x \,.\, A}{[t/x]A}\,\forall\mathrm{E} \end{array}} \Downarrow_s \mathbf{app}(e_1, e_2)}\;\mathrm{x}\forall\mathrm{E}$$

Figure 2.27: Program extraction/simplification, implication and universal quantification rules

$$\dfrac{\Gamma \vdash t \Downarrow^{i} e_1 \quad \mathrm{Inf}\,A \quad \langle\, \Gamma,\Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ {[t/x]A}\end{array}} \Downarrow_s e_2}{\langle\, \Gamma,\Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ \dfrac{[t/x]A}{\exists x\,.\,A}\,\exists\mathrm{I}\end{array}} \Downarrow_s \langle\, e_1, e_2 \,\rangle}\;\mathrm{xs}\exists\mathrm{I}
\qquad
\dfrac{\mathrm{Uninf}\,A \quad \Gamma \vdash t \Downarrow^{i} e}{\langle\, \Gamma,\Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ \dfrac{[t/x]A}{\exists x\,.\,A}\,\exists\mathrm{I}\end{array}} \Downarrow_s e}\;\mathrm{xs}\exists\mathrm{IL}$$

$$\dfrac{\mathrm{Inf}\,A \qquad \langle\, \Gamma,\Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ \exists x\,.\,A\end{array}} \Downarrow_s e_1 \qquad \langle\, (\Gamma, x \Downarrow^{i} x'), (\Delta, \boxed{\begin{array}{c}p\\ A\end{array}} \Downarrow_s p') \,\rangle \vdash \boxed{\begin{array}{c}p\\ A\\ \mathcal{Q}\\ C\end{array}} \Downarrow_s e_2}{\langle\, \Gamma,\Delta \,\rangle \vdash \boxed{\dfrac{\begin{array}{c}\mathcal{P}\\ \exists x\,.\,A\end{array} \quad \begin{array}{c}\overline{A}^{\,p}\\ \mathcal{Q}\\ C\end{array}}{C}\,\exists\mathrm{E}^{p}} \Downarrow_s \mathbf{spread}(e_1;\ x', p'\,.\,e_2)}\;\mathrm{xs}\exists\mathrm{E}$$

$$\dfrac{\mathrm{Uninf}\,A \qquad \langle\, \Gamma,\Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ \exists x\,.\,A\end{array}} \Downarrow_s e_1 \qquad \langle\, (\Gamma, x \Downarrow^{i} x'), \Delta \,\rangle \vdash \boxed{\begin{array}{c}p\\ A\\ \mathcal{Q}\\ C\end{array}} \Downarrow_s e_2}{\langle\, \Gamma,\Delta \,\rangle \vdash \boxed{\dfrac{\begin{array}{c}\mathcal{P}\\ \exists x\,.\,A\end{array} \quad \begin{array}{c}\overline{A}^{\,p}\\ \mathcal{Q}\\ C\end{array}}{C}\,\exists\mathrm{E}^{p}} \Downarrow_s \mathbf{app}((\mathbf{lam}\,x'\,.\,e_2), e_1)}\;\mathrm{xs}\exists\mathrm{EL}$$

$$\dfrac{\langle\, \Gamma,\Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ \bot\end{array}} \Downarrow_s e}{\langle\, \Gamma,\Delta \,\rangle \vdash \boxed{\begin{array}{c}P\\ \dfrac{\bot}{C}\,\bot\mathrm{E}\end{array}} \Downarrow_s \mathbf{any}(e)}\;\mathrm{xs}\bot$$

Figure 2.28: Program extraction/simplification, existential quantification and absurdity rules

$$\frac{\mathrm{Inf}\,A \qquad \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P}_1 \\ [\mathrm{zero}/x]A \end{array}} \Downarrow_s e_1 \qquad \langle\, (\Gamma, x \Downarrow^i x'), (\Delta, \boxed{\begin{array}{c} p \\ A \end{array}} \Downarrow_s p') \,\rangle \vdash \boxed{\begin{array}{c} p \\ A \\ \mathcal{P} \\ [\mathrm{succ}\ x/x]A \end{array}} \Downarrow_s e_2}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\dfrac{\begin{array}{c} \mathcal{P}_1 \\ [\mathrm{zero}/x]A \end{array} \qquad \begin{array}{c} \overline{A}^{\,p} \\ \mathcal{P} \\ [\mathrm{succ}\ x/x]A \end{array}}{\forall x \,.\, A}\,\mathrm{IND}^p} \Downarrow_s \mathbf{nat\_ind}(e_1;\ x', p' \,.\, e_2)}\ \mathrm{xsIND}$$

Figure 2.29: Program extraction/simplification, induction rule

The elimination rules for conjunction, implication, and existential quantification require checking for uninformative subformulas to avoid applying a destructor (projection, function application, or **spread**) inappropriately. For the elimination of an implication $A \supset B$, naive extraction yields $\mathbf{app}(e_1, e_2)$. But we must now account for the possibility that A is uninformative and thus e_1 is not a function; in that case the program to be extracted is e_1 alone. (If B is Harrop the rule xsH of course applies.) The conjunction elimination rules are treated in an analogous way. Naive extraction for the existential quantifier elimination rule yields an expression $\mathbf{spread}(e_1;\ x_1, x_2 \,.\, e_2)$ where x_1 is bound to the witness term and x_2 to the proof term during the evaluation of e_2. But if A is uninformative there is no proof term, and instead we extract $\mathbf{app}(\mathbf{lam}\, x_1 \,.\, e_2, e_1)$.

The rules xs⊃IR (Figure 2.27) and xs∃EL (Figure 2.28), which deal with implication introduction and existential elimination respectively, have the interesting feature that extraction is performed on subproofs containing proof variables for which no extraction assignment is introduced in the context. Thus for instance the rule xs⊃IR can succeed only if

$$\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} p \\ A \\ \mathcal{P} \\ B \end{array}} \Downarrow_s e$$

can be deduced *for arbitrary p.* At first glance it might seem that extraction could fail without an assignment for p. But for uninformative formulas extraction/simplification does not need to examine the proof. As a consequence we have the following:

Lemma 2.1 (Elimination of proofs of uninformative formulas) *If* Uninf A *and there is an extraction deduction*

$$\mathcal{E} :: \langle\, \Gamma, (\Delta_1, \boxed{\begin{array}{c} p \\ A \end{array}} \Downarrow_s p', \Delta_2) \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P} \\ B \end{array}} \Downarrow_s e$$

then there is an extraction deduction

$$\mathcal{E}' :: \langle\, \Gamma, (\Delta_1, \Delta_2) \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P} \\ B \end{array}} \Downarrow_s e$$

Proof 2.2 The proof is an easy induction on the structure of $\mathcal{E}$.

In the cases where $\mathcal{E}$ ends in xs¬, xs⊥, or xsH, the deduction does not depend on the environment.

If $\mathcal{E}$ ends in extraction from a proof variable:

$$\frac{(\Delta_1, \boxed{\begin{smallmatrix} p \\ A \end{smallmatrix}} \Downarrow_s p', \Delta_2)\left(\boxed{\begin{smallmatrix} q \\ B \end{smallmatrix}}\right) = q'}{\langle\, \Gamma, (\Delta_1, \boxed{\begin{smallmatrix} p \\ A \end{smallmatrix}} \Downarrow_s p', \Delta_2)\,\rangle \vdash \boxed{\begin{smallmatrix} q \\ B \end{smallmatrix}} \Downarrow q'}\ \textsf{ex-pvar}$$

there are two cases. If B is A then B is uninformative and there is an extraction deduction for B ending in xs¬, xs⊥, or xsH. Otherwise construct the deduction

$$\frac{(\Delta_1, \Delta_2)\left(\boxed{\begin{smallmatrix} q \\ B \end{smallmatrix}}\right) = q'}{\langle\, \Gamma, (\Delta_1, \Delta_2)\,\rangle \vdash \boxed{\begin{smallmatrix} q \\ B \end{smallmatrix}} \Downarrow q'}\ \textsf{ex-pvar}$$

The other cases are easy inductions since every assumption in the conclusion of an extraction deduction occurs in every subdeduction. □

In Figure 2.30, we show a fragment of the Elf implementation of simplifying extraction. There is one feature of this program that has not appeared before: some dependently-typed terms are annotated with their types. The following rule for extraction from an implication introduction is an example:

```
exs_implil : extract_simp (impliesi (P: |- A -> |- B)) M
             <- uninf A <- inf B
             <- {p: |- A} extract_simp (P p) M.
```

Here the argument P of **impliesi** is a function from a proof of some proposition A to a proof of some proposition B, and one of the premises is that A is uninformative. We annotate P with its type in order to express this premise.

Related work

The use of syntactic criteria to detect formulas void of computational content comes from the PX system [Hay90]. The idea is developed for the Calculus of Constructions by Paulin-Mohring [PM89] and for Nuprl by Sasaki [Sas86].

Both PX and Constructions are stronger logics than ours in which programs are terms of the logic and realizability is definable in the logic. This provides a different basis for proving the soundness of extraction.

Unlike our extraction and Paulin-Mohring's, extraction in PX does not reduce a functional type $A \rightarrow B$ where A is uninformative (type 0 in the terminology of PX) to the type B. As a result the types of extracted terms retain more structure than in our scheme.

```
extract_simp : |- A  -> term -> type.

exs_unit : extract_simp (P: |- A) unity <- harrop A.

exs_implii : extract_simp (impliesi (P: |- A -> |- B)) M
             <- uninf A <- inf B
             <- {p: |- A} extract_simp (P p) M.

exs_imple1 : extract_simp (impliese (P: |- (implies A B)) Q) M
             <- uninf A <- inf B <- extract_simp P M.

exs_andi2 : extract_simp (andi (P: |- A) (Q: |- B)) M
            <- uninf A <- inf B <- extract_simp Q M.

exs_andel1 : extract_simp (andel (P : |- (and A B))) M
             <- inf A <- uninf B <- extract_simp P M.

exs_ander1 : extract_simp (ander (P : |- (and A B))) M
             <- uninf A <- inf B <- extract_simp P M.

exs_existsi1 : extract_simp (existsi A T _) M
               <- ({x:i} uninf (A x)) <- extract_tm T M.

exs_existse1 : extract_simp (existse P (Q: |- (exists A))) (app (lam M) N)
               <- ({X:i} uninf (A X))
               <- ({X:i} {x:term} extract_tm X x
                   -> {p: |- (A X)} extract_simp (P X p) (M x))
               <- extract_simp Q N.

exs_existse2 : extract_simp (existse P_min P_maj) (spread N M)
               <- ({X:i} inf (A X))
               <- ({X:i} {x:term} extract_tm X x
                   -> {P:|- (A X)} {p:term} extract_simp P p
                   -> extract_simp (P_min X P) (M x p))
               <- extract_simp P_maj N.
```

Figure 2.30: A fragment of program extraction/simplification in Elf

In our system extraction is defined for all propositions and proofs, whether informative or not. In Constructions extraction is defined only for informative terms; but as Paulin-Mohring observes it can be defined for all terms essentially in the same way as in our system.

Chapter 3

Some Metatheory

Some correctness properties of extraction can be partially verified in Elf by formulating deduction transformations along the lines described in [PR92], [HP92]. As these authors point out, the verification technique is essentially that described in [Des86], but the dependent types of LF eliminate the need for explicit reasoning about the validity of the objects involved, and the term and type reconstruction of Elf mechanize the management of many details. We give informal proofs of the correctness properties along with their partial internalization in Elf; they cannot be completely internalized since there is no internally representable induction principle for LF signatures.

The first property of extraction we consider is *type soundness*: when we extract a program expression e from a proof of a proposition A, and we extract a type τ from A, we want to be able to deduce $e \in \tau$ in the type assignment system of Figure 2.10.

Theorem 3.1 (Type soundness of extraction) *For any $\mathcal{P}$, A, e, and τ, if* $\vdash \boxed{\begin{matrix}\mathcal{P}\\ A\end{matrix}} \Downarrow e$ *and* $\vdash A \Downarrow^t \tau$ *then* $\vdash e \in \tau$.

The second property we consider is *evaluation soundness*: when we extract a program e from a proof $\mathcal{P}$ of a proposition A, and e evaluates to v, we want there to be a proof $\mathcal{P}'$ of A from which we can extract v.

Theorem 3.2 (Evaluation soundness of extraction) *For any $\mathcal{P}$, A, e, and v, if* $\vdash \boxed{\begin{matrix}\mathcal{P}\\ A\end{matrix}} \Downarrow e$ *and* $\vdash e \hookrightarrow v$ *then there is a proof $\mathcal{P}'$ such that* $\vdash \boxed{\begin{matrix}\mathcal{P}'\\ A\end{matrix}} \Downarrow v$

Both type and evaluation soundness have proofs by straightforward induction over the structure of extraction and evaluation deductions, respectively. These proofs construct formal deductions of the required type from given formal deductions. The dual nature of Elf signatures – which can be viewed as either logic programs or language definitions – supports the direct expression of the constructive parts of the proofs. Each case of the induction is expressed as an Elf clause that matches a deduction of a part ılar shape. Thus to partially internalize the proof of type soundness we formulate an Elf sign: ıre to translate deductions of extraction judgments to deductions of typing assignment judgments. The signatures that define extraction and typing assignment judgments are viewed here as language definitions, although they are also executable Elf programs.

When discussing the proofs, we will need to name the formal deductions of extraction, typing, etc. To say that some judgment J is derivable, we often write simply "J"; we write $\mathcal{D} :: J$ when $\mathcal{D}$ is a deduction of the judgment J.

This chapter is organized as follows: first we present a proof of type soundness for naive extraction, with its partial formalization in Elf. Then we show how to modify the proof for extraction/simplification. A similar but briefer presentation of evaluation soundness follows. An interesting feature of the evaluation soundness proof is that its formalization implements a set of reductions for intuitionistic proofs with induction.

3.1 Type soundness

The proof is an induction on the structure of the extraction deduction. Since extraction deductions in general depend on a context of extraction assumptions, we generalize the theorem accordingly. The proof must construct a typing deduction for every extraction deduction, so it is necessary to define the typing assignments that correspond to a context of extraction assumptions. In accordance with the propositions-as-types principle, the typing assignment $p \in \tau$ that corresponds to an extraction assumption $\boxed{\begin{matrix}\mathcal{P}\\ A\end{matrix}} \Downarrow p$ depends only on the proposition A and not on the proof $\mathcal{P}$. So the following lemma allows us to translate any context of extraction assumptions to a context of typing assignments:

Lemma 3.3 (Totality of type extraction) *For any formula A there is a unique type τ such that $\vdash A \Downarrow^t \tau$.*

The proof is an easy induction on the structure of formulas, appealing to the type extraction deduction rules of Figure 2.13.

Definition 3.4 *Given a context of extraction assumptions $\langle \Gamma, \Delta \rangle$ where*
$\Gamma = x_1 \Downarrow^i x'_1, \ldots, x_m \Downarrow^i x'_m$ *and* $\Delta = \boxed{\begin{matrix}p_1\\ A_1\end{matrix}} \Downarrow p'_1, \ldots, \boxed{\begin{matrix}p_n\\ A_n\end{matrix}} \Downarrow p'_n$
we define

1. $\lceil \Gamma \rceil = x'_1 \in \mathsf{nat}, \ldots, x'_m \in \mathsf{nat}$
2. $\lceil \Delta \rceil = p'_1 \in \tau_1, \ldots, p'_n \in \tau_n$ *where* $A_1 \Downarrow^t \tau_1, \ldots, A_n \Downarrow^t \tau_n$.
3. $\lceil \langle \Gamma, \Delta \rangle \rceil = \lceil \Gamma \rceil, \lceil \Delta \rceil$, *i.e., the typing assignment context obtained by appending* $\lceil \Gamma \rceil$ *and* $\lceil \Delta \rceil$.

With these definitions we can generalize Theorem 3.1 to the following.

Lemma 3.5 (Type soundness of extraction in arbitrary contexts)
If $\langle \Gamma, \Delta \rangle \vdash \boxed{\begin{matrix}\mathcal{P}\\ A\end{matrix}} \Downarrow e$ *and* $\vdash A \Downarrow^t \tau$ *then* $\lceil \langle \Gamma, \Delta \rangle \rceil \vdash e \in \tau$.

Since the extraction judgment depends on the judgment $t \Downarrow^{i} e$ (extraction from individual terms) a type soundness property for individual term extraction is also needed:

Lemma 3.6 (Type soundness for individual extraction) *If* $\Gamma \vdash t \Downarrow^{i} e$ *then* $\lceil \Gamma \rceil \vdash e \in \mathsf{nat}$.

The proof is trivial since the only terms in the object logic are natural numbers. Its representation in Elf is a very simple example of the partial verification technique that we apply to the proof of Lemma 3.5. Here is the Elf signature that represents the proof of Lemma 3.6:

```
tsetm : extract_tm T M -> of M nat -> type.

tsetm_zero : tsetm ex_zero tp_0.
tsetm_succ : tsetm (ex_succ E) (tp_s D) <- tsetm E D.
```

The clause `tsetm_zero` corresponds to the base case of the proof, which constructs the depth-one typing derivation for the program expression 0. The clause `tsetm_succ` contains a subgoal that represents an appeal to the induction hypothesis. There is no clause translating extraction from variables (rule ex-ivar of Figure 2.15), because extraction contexts are represented at the meta-level as Elf contexts. This is reflected by representing the translation by a meta-level assumption, a technique shown in the presentation to follow.

The encoding of the proof of Lemma 3.5 in Elf follows the same basic principles. We declare a type family to represent the lemma:

```
tse : extract (P: |- A) M -> extract_tp A T -> of M T -> type.
```

This declaration expresses a relation between a program extraction deduction, a type extraction deduction, and a typing assignment deduction. Proving Lemma 3.5 amounts to giving a total function from the first two deductions to the third. The encoding of the proof in Elf is partial in the following sense: we can code this function as an Elf signature, and the well-typedness of each declaration in the signature guarantees that the construction carried out is correct. But there is no internal guarantee that the signature determines a total function (and clearly we cannot code the construction directly as an Elf function since it is not schematic). This guarantee could probably be obtained by applying the *schema checking* of [PR92] to the proofs of type soundness, and possibly also to evaluation soundness. We have partially schema-checked the type soundness proof by hand. Further work in this direction is beyond the scope of the thesis since full schema checking has not been implemented.

We give some characteristic cases of the proof. Reference to Figures 2.10, 2.11 (type assignment), 2.13 (type extraction), and 2.14 through 2.19 will help in following the argument. Since the extraction deduction $\mathcal{E} :: \langle \Gamma, \Delta \rangle \vdash \boxed{\begin{matrix}\mathcal{P}\\ A\end{matrix}} \Downarrow e$ closely follows the structure of the object proof $\mathcal{P}$ the cases can be characterized by the last inference of $\mathcal{P}$. The cases are of the following main kinds: base cases ($\mathcal{P}$ ends in $\top$I or one of the axiom schemas of arithmetic), *first-order* inductive cases where the last inference does not bind any individuals or discharge any assumptions ($\wedge$I, $\wedge$E$_L$, $\wedge$E$_R$, $\vee$I$_L$, $\vee$I$_R$, $\supset$E, and $\neg$E), and *higher-order* inductive cases where the last inference binds a parameter and/or discharges an assumption ($\vee$E, $\supset$I, $\neg$I, $\forall$I, $\exists$E, $\exists$I, $\forall$E, and IND). The lemmas needed for each type of case are introduced as needed. A simple lemma needed throughout the proof is:

Lemma 3.7 (Inversion for type extraction) *Given $\mathcal{E} :: \vdash A \Downarrow^t \tau$, the last inference rule of $\mathcal{E}$ is uniquely determined by the form of A.*

The proof is easily seen by inspecting the rules of Figure 2.13.

Base cases

The base cases are those in which the extraction deduction $\mathcal{E} :: \langle\, \Gamma, \Delta\, \rangle \vdash \boxed{\begin{matrix}\mathcal{P}\\ A\end{matrix}} \Downarrow e$ consists of the application of a single rule without premises. For our simple system of arithmetic, these are the rules for extraction from a proof of $\top$, an equality, or the negation of an equality. We show the reasoning for the negative axiom schema AXO.

Case $\mathcal{E}$ =

$$\dfrac{}{\langle\, \Gamma, \Delta\, \rangle \vdash \boxed{\dfrac{}{\neg \text{succ}\ t = \text{zero}}\text{AXO}} \Downarrow \textbf{neg}}\text{XAXO}$$

By inversion any type extraction from $\neg$succ t = zero has the form

$$\dfrac{\dfrac{}{\text{succ}\ t = \text{zero} \Downarrow^t \textsf{atom}}\textsf{xt=}}{\neg\text{succ}\ t = \text{zero} \Downarrow^t \textsf{atom} \Rightarrow \textsf{void}}\textsf{xt}\neg$$

Then construct the following typing deduction of the form required:

$$\dfrac{}{\lceil\langle\, \Gamma, \Delta\, \rangle\rceil \vdash \textbf{neg} \in \textsf{atom} \Rightarrow \textsf{void}}\textsf{tp-neg}$$

□

The constructive content of this case is expressed by the Elf clause:

```
tse_ax_zero : tse ex_ax_zero
                (ext_not (Et: extract_tp (eq (succ T) zero) atom))
                (tp_neg atom).
```

First-order inductive cases

For cases where the object proof $\mathcal{P}$ ends in one of the rules $\wedge$I, $\wedge E_L$, $\wedge E_R$, $\vee I_L$, $\vee I_R$, $\supset$E, or $\neg$E, no extraction assumptions are introduced into the context in the subdeduction(s) of $\mathcal{E}$. These cases are straightforward. We apply the totality of type extraction to obtain type extraction deductions corresponding to the subdeductions. We can then apply the induction hypothesis to the subdeductions. We show the case for $\supset$E as an example.

Case $\mathcal{E}$ =

$$\dfrac{\begin{matrix}\mathcal{E}_1\\ \langle\,\Gamma,\Delta\,\rangle \vdash \boxed{\begin{matrix}\mathcal{P}_1\\ A\supset B\end{matrix}} \Downarrow e_1\end{matrix} \qquad \begin{matrix}\mathcal{E}_2\\ \langle\,\Gamma,\Delta\,\rangle\vdash \boxed{\begin{matrix}\mathcal{P}_2\\ A\end{matrix}}\Downarrow e_2\end{matrix}}{\langle\,\Gamma,\Delta\,\rangle \vdash \boxed{\dfrac{\begin{matrix}\mathcal{P}_1\\ A\supset B\end{matrix}\qquad\begin{matrix}\mathcal{P}_2\\ A\end{matrix}}{B}\,{\supset}\mathrm{E}} \Downarrow \mathbf{app}(e_1,e_2)}\,\mathsf{x{\supset}E}$$

Assume there is a type extraction deduction $\mathcal{T} :: B \Downarrow^t \tau$.

Since type extraction is total there is a type extraction $\mathcal{T}' :: A \Downarrow^t \tau'$. Construct the type extraction $\mathcal{T}''$ =

$$\dfrac{\begin{matrix}\mathcal{T}'\\ A\Downarrow^t \tau'\end{matrix}\qquad\begin{matrix}\mathcal{T}\\ B\Downarrow^t \tau\end{matrix}}{A\supset B \Downarrow^t \tau' \Rightarrow \tau}\,\mathsf{xt{\supset}}$$

Applying the induction hypothesis to $\mathcal{E}_1$ and $\mathcal{T}''$ yields the typing deduction $\mathcal{D}_1 :: \lceil\langle\,\Gamma,\Delta\,\rangle\rceil \vdash e_1 \in \tau' \Rightarrow \tau$. Applying it to $\mathcal{E}_2$ and $\mathcal{T}'$ yields $\mathcal{D}_2 :: \lceil\langle\,\Gamma,\Delta\,\rangle\rceil \vdash e_2 \in \tau'$. Then construct the typing deduction

$$\dfrac{\begin{matrix}\mathcal{D}_1\\ \lceil\langle\,\Gamma,\Delta\,\rangle\rceil \vdash e_1 \in \tau' \Rightarrow \tau\end{matrix}\qquad\begin{matrix}\mathcal{D}_2\\ \lceil\langle\,\Gamma,\Delta\,\rangle\rceil \vdash e_2 \in \tau'\end{matrix}}{\lceil\langle\,\Gamma,\Delta\,\rangle\rceil \vdash \mathbf{app}(e_1,e_2) \in \tau}\,\mathsf{tp\text{-}app}$$

□

The Elf clause representing this case is:

```
tse_impliese : tse (ex_impliese E2 E1) Et (tp_app D2 D1)
               <- tse E1 (ext_implies Et Et') D1
               <- tse E2 Et' D2.
```

The two subgoals represent the two uses of the induction hypothesis. The term `(ext_implies Et Et')` represents the construction of the type extraction $\mathcal{T}''$.

Higher-order inductive cases

Where the object proof $\mathcal{P}$ ends in a rule involving a binding construct, some manipulation of the context is required. When t[illegible]y discharged assumptions and no bound individuals ($\vee$E, $\supset$I, $\neg$I), Lemma 3.7 (inversion) and Definition 3.4 (context translation) are sufficient for the application of the induction hypothesis. We show the case for $\supset$I as an example of this kind of reasoning.

Case $\mathcal{E}$ =

$$\dfrac{\begin{array}{c}\mathcal{E}' \\ \langle\, \Gamma, (\Delta, \boxed{\begin{array}{c}p\\A\end{array}} \Downarrow p')\,\rangle \vdash \boxed{\begin{array}{c}p\\A\\\mathcal{P}\\B\end{array}} \Downarrow e\end{array}}{\langle\, \Gamma, \Delta\,\rangle \vdash \boxed{\dfrac{\begin{array}{c}\overline{}^{\,p}\\A\\\mathcal{P}\\B\end{array}}{A \supset B}\supset\mathrm{I}^p} \Downarrow \mathbf{lam}\, p'.\, e}\ \mathsf{x}\supset\mathsf{I}$$

By inversion the corresponding type extraction deduction has the following form:

$$\dfrac{\begin{array}{c}\mathcal{T}_1\\ A \Downarrow^t \tau_1\end{array} \qquad \begin{array}{c}\mathcal{T}_2\\ B \Downarrow^t \tau_2\end{array}}{A \supset B \Downarrow^t \tau_1 \Rightarrow \tau_2}\ \mathsf{xt}\supset$$

By definition 3.4, $\lceil\langle\, \Gamma, (\Delta, \boxed{\begin{array}{c}p\\A\end{array}} \Downarrow p')\,\rangle\rceil = \lceil\Gamma\rceil, \lceil\Delta\rceil, p' \in \tau_1$.

Applying the induction hypothesis to $\mathcal{E}'$ with $\mathcal{T}_2$ yields the typing deduction $\mathcal{D} :: \lceil\Gamma\rceil, \lceil\Delta\rceil, p' \in \tau_1 \vdash e \in \tau_2$. Then construct:

$$\dfrac{\begin{array}{c}\mathcal{D}\\ \lceil\Gamma\rceil, \lceil\Delta\rceil, p' \in \tau_1 \vdash e \in \tau_2\end{array}}{\lceil\Gamma\rceil, \lceil\Delta\rceil \vdash \mathbf{lam}\, p'.\, e \in \tau_1 \Rightarrow \tau_2}\ \mathsf{tp\text{-}lam}$$

This is the typing derivation required since $\lceil\Gamma\rceil, \lceil\Delta\rceil = \lceil\langle\, \Gamma, \Delta\,\rangle\rceil$.
□

The Elf clause corresponding to this case is:

```
tse_impliesi : tse (ex_impliesi E) (ext_implies Et2 Et1) (tp_lam D)
                <- ({p} {p'} {e: extract p p'} {d} tse e Et1 d
                    -> tse (E p p' e) Et2 (D p' d)).
```

The term `(ext_implies Et2 Et1)` represents the use of the inversion principle. Since the extraction $\mathcal{E}'$ is represented as a term (E) of functional type, the appeal to the induction hypothesis is represented as a subgoal of higher-order judgment type. The type soundness assumption in this subgoal `tse e Et1 d` represents the translation of the assumption $\boxed{\begin{array}{c}p\\A\end{array}} \Downarrow p'$ in the extraction context to the typing context $p' \in \tau_1$, reflecting Definition 3.4.

When individual parameter binding is involved, as in $\forall$I and $\exists$E, we need a permutation property for typing assignment contexts:

Lemma 3.8 (Permutation for typing assignment contexts) *If* $\Gamma_1, x_1 \in \tau_1, \Gamma_2, x_2 \in \tau_2, \Gamma_3 \vdash e \in \tau$ *then* $\Gamma_1, x_2 \in \tau_2, \Gamma_2, x_1 \in \tau_1, \Gamma_3 \vdash e \in \tau$.

The lemma can be proved by an easy induction on the structure of typing derivations, keeping in mind the assumption that no variable occurs more than once in a context.

Here is the reasoning for type soundness for $\forall$I. It has nearly the same structure as the $\supset$I case.

Case $\mathcal{E} =$

$$\dfrac{\begin{array}{c}\mathcal{E}' \\ \langle\, (\Gamma, x \Downarrow^i x'), \Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P} \\ A\end{array}} \Downarrow e\end{array}}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P} \\ \dfrac{A}{\forall x\,.\,A}\,\forall\mathrm{I}\end{array}} \Downarrow \mathbf{lam}\, x'\,.\,e}\;\mathsf{x}\forall\mathsf{I}$$

By inversion the corresponding type extraction deduction has the following form:

$$\dfrac{\begin{array}{c}\mathcal{T}' \\ A \Downarrow^t \tau\end{array}}{\forall x\,.\,A \Downarrow^t \mathsf{nat} \Rightarrow \tau}\;\mathsf{xt}\forall$$

By definition 3.4, $\lceil \langle\, (\Gamma, x \Downarrow^i x'), \Delta \,\rangle \rceil = \lceil\Gamma\rceil, x' \in \mathsf{nat}, \lceil\Delta\rceil$.

Applying the induction hypothesis to $\mathcal{E}'$ with $\mathcal{T}'$ yields the typing deduction $\mathcal{D} :: \lceil\Gamma\rceil, x' \in \mathsf{nat}, \lceil\Delta\rceil \vdash e \in \tau$. We apply the permutation lemma to obtain a typing deduction $\mathcal{D}' :: \lceil\Gamma\rceil, \lceil\Delta\rceil, x' \in \mathsf{nat} \vdash e \in \tau$.

Then construct:

$$\dfrac{\begin{array}{c}\mathcal{D}' \\ \lceil\Gamma\rceil, \lceil\Delta\rceil, x' \in \mathsf{nat} \vdash e \in \tau\end{array}}{\lceil\Gamma\rceil, \lceil\Delta\rceil \vdash \mathbf{lam}\, x'\,.\,e \in \mathsf{nat} \Rightarrow \tau}\;\mathsf{tp\text{-}lam}$$

□

The Elf clause implementing this case is:

```
tse_foralli : tse (ex_foralli E) (ext_forall Et) (tp_lam D)
               <- ({x} {x'} {e:extract_tm x x'} {d}
                  tsetm e d -> tse (E x x' e) (Et x) (D x' d)).
```

Again we use a subgoal of higher-order judgment type to represent the use of the induction hypothesis. In this case the context translation is represented by the assumption `tsetm e d`, since the context is extended in its individual variable domain rather than its proof variable domain. The use of the permutation lemma is not explicit, because contexts are represented as LF contexts, for which permutation holds.

The rules $\exists$I, $\forall$E, and IND involve substitution in propositions. These cases use the property that substitution of individuals has no effect on type extractions:

Lemma 3.9 Substitution lemma for type extraction: *If* $\vdash A \Downarrow^t \tau$ *then* $\vdash [t/x]A \Downarrow^t \tau$.

The proof is quite simple, but requires a definition of substitution for individual variables in type extraction deductions. For any $\mathcal{T} :: A \Downarrow^t \tau$, the substitution of a term t for an individual variable x in $\mathcal{T}$ is constructed by substituting t for x in A, and recursively substituting t for x in any subdeductions of $\mathcal{T}$. It is easy to see that this results in a valid type extraction $\mathcal{T}' :: [t/x]A \Downarrow^t \tau$.

In addition the extraction rules for $\exists$I and $\forall$E depend on extractions from individual terms of the logic, so these cases use Lemma 3.6 (type soundness for extraction from individuals).

We give the reasoning for the rule of existential introduction:

Case $\mathcal{E}$ =

$$\frac{\begin{array}{c}\mathcal{E}_1\\ \Gamma \vdash t \Downarrow^i e_1\end{array} \qquad \begin{array}{c}\mathcal{E}_2\\ \langle \Gamma, \Delta \rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ [t/x]A\end{array}} \Downarrow e_2\end{array}}{\langle \Gamma, \Delta \rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ \dfrac{[t/x]A}{\exists x\,.\,A}\,\exists\mathrm{I}\end{array}} \Downarrow \langle e_1, e_2 \rangle}\ \mathsf{x}\exists\mathsf{I}$$

By inversion, the corresponding type extraction deduction has the form

$$\frac{\begin{array}{c}\mathcal{T}_1\\ A \Downarrow^t \tau'\end{array}}{\exists x\,.\,A \Downarrow^t \mathsf{nat} \times \tau'}\ \mathsf{xt}\exists$$

Applying Lemma 3.9 (substitution in type extractions) to $\mathcal{T}_1$, we obtain a type extraction $\mathcal{T}_2 :: [t/x]A \Downarrow^t \tau'$. Lemma 3.6 applied to $\mathcal{E}_1$ gives the typing assignment $\mathcal{D}_1 :: \lceil \langle \Gamma, \Delta \rangle \rceil \vdash e_1 \in \mathsf{nat}$. Then the induction hypothesis applied to $\mathcal{E}_2$ with $\mathcal{T}_2$ yields the typing assignment $\mathcal{D}_2 :: \lceil \langle \Gamma, \Delta \rangle \rceil \vdash e_2 \in \tau'$, and thus

$$\frac{\begin{array}{c}\mathcal{D}_1\\ \lceil \langle \Gamma, \Delta \rangle \rceil \vdash e_1 \in \mathsf{nat}\end{array} \qquad \begin{array}{c}\mathcal{D}_2\\ \lceil \langle \Gamma, \Delta \rangle \rceil \vdash e_2 \in \tau'\end{array}}{\lceil \langle \Gamma, \Delta \rangle \rceil \vdash \langle e_1, e_2 \rangle \in \mathsf{nat} \times \tau'}\ \mathsf{tp\text{-}pr}$$

□

The Elf clause for this case is:

```
tse_existsi : tse (ex_existsi E2 (E1:extract_tm T M))
                 (ext_exists Et) (tp_pair D2 D1)
              <- tsetm E1 D1
              <- tse E2 (Et T) D2.
```

The term (Et T) represents the use of the substitution lemma. This representation is justified by a general property (called *compositionality* in [HHP93]) of our encodings. Compositionality guarantees that representations are faithful with respect to substitution. In particular, [T/x]E represents $[t/x]e$ if T represents t and E represents e (where the parameter x is identified with the variable x). Recall that the higher-order abstract syntax encoding of $\exists$ represents $\exists x \,.\, A$ as a term of the form (exists A) where A has functional type. Thus the Elf representation of the type extraction rule xt$\exists$ contains a premise of (dependent) functional type as well:

```
ext_exists : extract_tp (exists A) (* nat Tp)
              <- {x:i} extract_tp (A x) Tp.
```

So the representation of the type extraction deduction $\mathcal{T}_1$ is not Et but (Et x) for some parameter x, and we obtain the substitution [T/x](Et x) by β-reduction of the term (Et T).

3.2 Type soundness for extraction/simplification

A type soundness theorem for the extraction/simplification judgment $\Downarrow_s$ is more valuable, as the property is no longer so obvious. We present the proof as a modification of the proof of the previous section.

Theorem 3.10 (Type soundness of extraction/simplification) *If* $\vdash \boxed{\begin{matrix}\mathcal{P}\\ A\end{matrix}} \Downarrow_s e$ *and* $\vdash A \Downarrow_s^t \tau$ *then* $\vdash e \in \tau$.

Again we generalize the theorem to arbitrary contexts, so a translation of extraction assumption contexts to typing assignment contexts is needed. The required definition is the obvious modification of Definition 3.4. For the translation to be well-defined, we need the analog of Lemma 3.3, totality of type extraction.

Lemma 3.11 (Totality of type extraction/simplification) *For any formula A there is a unique type τ such that* $\vdash A \Downarrow_s^t \tau$.

The proof is an easy induction on the structure of formulas, appealing to the type extraction deduction rules of Figure 2.23, together with the following (easily seen by inspection of Figures 2.20 through 2.22):

Lemma 3.12 *The judgment* Inf *is the complement of the judgment* Uninf, *and for any formula A, if* Harrop A *is deducible then* Uninf A *is deducible.*

Definition 3.13 *Given a context of extraction assumptions* $\langle\, \Gamma, \Delta \,\rangle$ *where*
$\Gamma = x_1 \Downarrow^i x_1', \ldots, x_m \Downarrow^i x_m'$ *and* $\Delta = \boxed{\begin{matrix}p_1\\ A_1\end{matrix}} \Downarrow_s p_1', \ldots, \boxed{\begin{matrix}p_n\\ A_n\end{matrix}} \Downarrow_s p_n'$
we define

i. $\lceil \Gamma \rceil = x_1' \in \mathsf{nat}, \ldots, x_m' \in \mathsf{nat}$

ii. $\lceil\Delta\rceil = p'_1 \in \tau_1, \ldots, p'_n \in \tau_n$ *where* $A_1 \Downarrow^t_s \tau_1, \ldots, A_n \Downarrow^t_s \tau_n$.

iii. $\lceil\langle\, \Gamma, \Delta\, \rangle\rceil = \lceil\Gamma\rceil, \lceil\Delta\rceil$, *i.e., the typing assignment context obtained by appending* $\lceil\Gamma\rceil$ *and* $\lceil\Delta\rceil$.

Using these definitions we generalize the type soundness theorem as before:

Lemma 3.14 (Type soundness of extraction/simplification in arbitrary contexts) *If* $\langle\, \Gamma, \Delta\, \rangle \vdash \boxed{\begin{matrix}\mathcal{P}\\ A\end{matrix}} \Downarrow_s e$ *and* $\vdash A \Downarrow^t_s \tau$ *then* $\lceil\langle\, \Gamma, \Delta\, \rangle\rceil \vdash e \in \tau$.

Extraction/simplification depends on the same individual term extraction judgment $\Downarrow^i$ that naive extraction does, thus Lemma 3.6 (type soundness for term extraction) is used in the new proof.

We appeal to an inversion principle for type extraction/simplification throughout the proof, a modification of Lemma 3.7.

Lemma 3.15 (Inversion for type extraction/simplification) *Given* $\mathcal{E} :: A \Downarrow^t_s \tau$, *the last inference rule of* $\mathcal{E}$ *is uniquely determined by the form of* A.

This is proved easily by inspection of the inference system of Figure 2.23, keeping in mind Lemma 3.12.

Again we present a few characteristic cases of the induction on the structure of the given extraction deduction $\mathcal{E}$. Reference to Figures 2.10, 2.11, 2.20 through 2.27, and 2.30 will help in following the presentation.

When the extraction deduction ends in a rule that does no simplification (all subformulas are informative) the proof for naive extraction carries over directly. Thus we do not show the reasoning for deductions ending in xs∧I, xs∧E$_L$, xs∧E$_R$, xs⊃I, xs⊃E, x∀I, x∀E, xs∃I, xs∃E, or xsIND of Figures 2.25 through 2.29. Of the extractions that perform simplifications, we again have base cases, first-order inductive cases, and higher-order inductive cases.

Base cases

The base cases concern extraction from uninformative proofs, where the end formula is a Harrop formula, a negation, or $\bot$. The first two of these are trivial, constructing depth-one typing derivations corresponding to the depth-one extraction derivations.

Although the reasoning is easy it is worth examining the case for extraction from proofs of $\bot$ since it has no analog in the proof for naive extraction.

Case $\mathcal{E} =$

$$\frac{}{\langle\, \Gamma, \Delta\, \rangle \vdash \boxed{\begin{matrix}\mathcal{P}\\ \bot\end{matrix}} \Downarrow_s \mathbf{app}(\mathbf{neg}, ())}\ \mathsf{xs}\bot$$

Assuming the type extraction $\mathcal{T} :: \bot \Downarrow^t_s \tau$, the inversion principle gives $\tau = \mathsf{void}$. Then construct

$$\frac{\dfrac{}{\lceil\langle\, \Gamma, \Delta\, \rangle\rceil \vdash \mathbf{neg} \in \mathsf{unit} \Rightarrow \mathsf{void}}\ \mathsf{tp\text{-}neg} \qquad \dfrac{}{\lceil\langle\, \Gamma, \Delta\, \rangle\rceil \vdash () \in \mathsf{unit}}\ \mathsf{tp\text{-}unit}}{\lceil\langle\, \Gamma, \Delta\, \rangle\rceil \vdash \mathbf{app}(\mathbf{neg}, ()) \in \mathsf{void}}\ \mathsf{tp\text{-}app}$$

□

The Elf clause for this case:

```
    tse_f : tses exs_f exts_false (tp_app tp_unity (tp_neg unit)).
```

First-order inductive cases

When the object proof ends in one of the inference rules $\wedge$I, $\wedge E_L$, $\wedge E_R$, $\vee I_L$, $\vee I_R$, $\supset$E, or $\neg$E, and one of the subproofs is uninformative, the extraction rule performs simplification. As in the proof for naive extraction, these cases are straightforward because no assumptions are introduced into the context. Verifying them simply requires checking that the type extraction/simplification rules and the program extraction/simplification rules match up correctly, dropping the appropriate parts of the types and program expressions. As an example we show the case for $\supset$E.

Case $\mathcal{E}$ =

$$\dfrac{\overset{\mathcal{U}}{\text{Uninf}\,A} \quad \overset{\mathcal{I}}{\text{Inf}\,B} \quad \overset{\mathcal{E}'}{\langle \Gamma, \Delta \rangle \vdash \boxed{\begin{matrix}\mathcal{P}_1\\ A \supset B\end{matrix}} \Downarrow_s e}}{\langle \Gamma, \Delta \rangle \vdash \boxed{\dfrac{\begin{matrix}\mathcal{P}_1\\ A \supset B\end{matrix} \quad \begin{matrix}\mathcal{P}_2\\ A\end{matrix}}{B}\supset\text{E}} \Downarrow_s e}\,\text{xs}\supset\text{ER}$$

Assume there is a type extraction deduction $\mathcal{T} :: B \Downarrow_s^t \tau$.

Construct the type extraction $\mathcal{T}'$ =

$$\dfrac{\overset{\mathcal{U}}{\text{Uninf}\,A} \qquad \overset{\mathcal{I}}{\text{Inf}\,B} \quad \overset{\mathcal{T}}{B \Downarrow_s^t \tau}}{A \supset B \Downarrow_s^t \tau}\,\text{xst}\supset\text{R}$$

Applying the induction hypothesis to $\mathcal{E}'$ and $\mathcal{T}'$ yields the typing deduction $\mathcal{D} :: \lceil\langle \Gamma, \Delta \rangle\rceil \vdash e \in \tau$ as required.

□

The Elf clause that implements this case follows.

```
    tse_impliese1 : tses (exs_imple1 E' Ib Ua) T D
                     <- tses E' (exts_impr T Ib Ua) D.
```

Higher-order inductive cases

Of the higher-order cases, only those for object proofs ending in $\supset$I, $\exists$I, and $\exists$E involve simplifications other than extraction of (). If A is uninformative, to extract a program from a proof of $\exists x\,.\,A$ by $\exists$I we extract just an expression e representing the witness t for which $[t/x]A$ holds, instead of a pair. As a result the proof for this case is simpler than its counterpart for the case where A is informative, since there is no need to reason about the type extracted from $[t/x]A$. In fact although the representation of the object proof is higher-order, the construction for this case is represented using only first-order subgoals.

Case $\mathcal{E}$ =

$$\dfrac{\overset{\mathcal{U}}{\text{Uninf}\,A} \qquad \overset{\mathcal{E}'}{\Gamma \vdash t \Downarrow^{i} e}}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ \dfrac{[t/x]A}{\exists x\,.\,A}\,\exists\text{I}\end{array}} \Downarrow_s e}\;\text{xs}\exists\text{IL}$$

The type extraction from $\exists x\,.\,A$ must have the form:

$$\dfrac{\overset{\mathcal{U}}{\text{Uninf}\,A}}{\exists x\,.\,A \Downarrow^{t}_{s} \textsf{nat}}\;\text{xst}\exists\text{L}$$

By Lemma 3.6 (type soundness for term extraction) there is a typing derivation $\mathcal{D} :: \lceil\langle\, \Gamma, \Delta \,\rangle\rceil \vdash e \in \textsf{nat}$ as required.
$\square$

The Elf clause that expresses this case is:

```
tse_existsil : tses (exs_existsil (Etm:extract_tm T M) Ua)
               (exts_exl Ua) D
               <- tsetm Etm D.
```

Next we consider cases (xs$\supset$IR or xs$\exists$EL) where the object proof discharges an uninformative assumption and as a result the extracted program is simplified. (Uninformative assumptions can also be discharged by $\vee$E and IND, but they do not affect the extraction: in the case of the induction rule IND, if the discharged assumption is uninformative, so is the end-formula of the proof, so the extraction rules for Harrop and negative formulas cover this case. In the case of $\vee$E where the major premise is $A \vee B$ with both A and B uninformative, extraction nevertheless retains the information of which holds, A or B.)

When extracting from proofs ending in $\supset$I or $\exists$E the uninformative assumptions do not lead to the introduction of assumptions in the extraction deduction (see the discussion in Section 2.3.2 and the rules in Figure 2.27 and Figure 2.28). As a result the reasoning for $\supset$I does not involve any extension of the context:

Case $\mathcal{E}$ =

$$\frac{\overset{\mathcal{U}}{\text{Uninf}\,A} \qquad \overset{\mathcal{I}}{\text{Inf}\,B} \qquad \overset{\mathcal{E}'}{\langle \Gamma,\Delta \rangle \vdash \boxed{\begin{array}{c} p \\ A \\ \mathcal{P} \\ B \end{array}} \Downarrow_s e}}{\langle \Gamma,\Delta \rangle \vdash \boxed{\begin{array}{c} \overline{\quad}^{\,p} \\ A \\ \mathcal{P} \\ \dfrac{B}{A \supset B}\supset\text{I}^p \end{array}} \Downarrow_s e}\ \text{xs}\supset\text{IR}$$

The corresponding type extraction must have the form:

$$\frac{\overset{\mathcal{U}}{\text{Uninf}\,A} \qquad \overset{\mathcal{I}}{\text{Inf}\,B} \qquad \overset{\mathcal{T}}{B \Downarrow_s^t \tau}}{A \supset B \Downarrow_s^t \tau}\ \text{xst}\supset\text{R}$$

By the induction hypothesis applied to $\mathcal{E}'$ and $\mathcal{T}$ there is a typing derivation $\mathcal{D} :: e \in \tau$ as required.
□

The following Elf rule formalizes this case.

```
tse_impliesil : tses (exs_implil E Ib Ua) (exts_impr Etb Ib Ua) D
                 <- ({p} tses (E p) Etb D).
```

Although the subgoal introduces the parameter p it does not introduce any assumptions on p. This reflects the fact that p stands for an uninformative proof whose structure cannot affect the extraction (E p) (Lemma 2.1 of Section 2.3.2).

Extraction for $\exists$E extends the context, but only in its individual variable domain.

Case $\mathcal{E}$ =

$$\frac{\overset{\mathcal{U}}{\text{Uninf}\,A} \qquad \overset{\mathcal{E}_1}{\langle \Gamma,\Delta \rangle \vdash \boxed{\begin{array}{c} \mathcal{P} \\ \exists x\,.\,A \end{array}} \Downarrow_s e_1} \qquad \overset{\mathcal{E}_2}{\langle (\Gamma, x \Downarrow^i x'),\Delta \rangle \vdash \boxed{\begin{array}{c} p \\ A \\ \mathcal{Q} \\ C \end{array}} \Downarrow_s e_2}}{\langle \Gamma,\Delta \rangle \vdash \boxed{\dfrac{\begin{array}{cc} & \overline{\quad}^{\,p} \\ & A \\ \mathcal{P} & \mathcal{Q} \\ \exists x\,.\,A & C \end{array}}{C}\exists\text{E}^p} \Downarrow_s \mathbf{app}((\mathbf{lam}\,x'\,.\,e_2), e_1)}\ \text{xs}\exists\text{EL}$$

Assume $\mathcal{T}_2 :: \vdash C \Downarrow_s^t \tau$.

Construct the type extraction derivation $\mathcal{T}_1$ =

$$\cfrac{\begin{array}{c}\mathcal{U}\\ \mathrm{Uninf}\, A\end{array}}{\exists x\, .\, A \Downarrow^{t}_{s} \mathsf{nat}}\ \mathsf{xst\exists L}$$

By the induction hypothesis applied to $\mathcal{E}_1$ and $\mathcal{T}_1$, there is a typing deduction $\mathcal{D}_1 :: \lceil\langle\, \Gamma, \Delta \,\rangle\rceil \vdash e_1 \in \mathsf{nat}$.

By the induction hypothesis applied to $\mathcal{E}_2$ and $\mathcal{T}_2$, there is a deduction $\mathcal{D}_2 :: \lceil\langle\, (\Gamma, x \Downarrow^{i} x'), \Delta \,\rangle\rceil \vdash e_2 \in \tau$. By definition $\lceil\langle\, (\Gamma, x \Downarrow^{i} x'), \Delta \,\rangle\rceil = \lceil\Gamma\rceil, x' \in \mathsf{nat}, \lceil\Delta\rceil$. By permutation for typing assignment contexts, there is a deduction $\mathcal{D}_2' :: \lceil\Gamma\rceil, \lceil\Delta\rceil, x' \in \mathsf{nat} \vdash e_2 \in \tau$. Then construct

$$\cfrac{\cfrac{\begin{array}{c}\mathcal{D}_2'\\ \lceil\langle\, \Gamma, \Delta \,\rangle\rceil, x' \in \mathsf{nat} \vdash e_2 \in \tau\end{array}}{\lceil\langle\, \Gamma, \Delta \,\rangle\rceil \vdash \mathbf{lam}\, x'\, .\, e_2 \in \mathsf{nat} \Rightarrow \tau}\ \mathsf{tp\text{-}lam} \qquad \begin{array}{c}\mathcal{D}_1\\ \lceil\langle\, \Gamma, \Delta \,\rangle\rceil \vdash e_1 \in \mathsf{nat}\end{array}}{\lceil\langle\, \Gamma, \Delta \,\rangle\rceil \vdash \mathbf{app}(\mathbf{lam}\, x'\, .\, e_2, e_1) \in \tau}\ \mathsf{tp\text{-}app}$$

□

This case is formalized by the following Elf clause.

```
tse_existsel : tses (exs_existsel Ee Ec U) Et
                   (tp_app D1 (tp_lam D2))
               <- tses Ee (exts_exl U) D1
               <- ({x} {x'} {etm:extract_tm x x'}
                     {d} tsetm etm d ->
                     {p} tses (Ec x x' etm p) Et (D2 x' d)).
```

As in the previous case the last subgoal introduces the uninformative proof parameter p without introducing assumptions on p.

3.3 Evaluation Soundness

The proof of evaluation soundness for naive extraction is an induction on the structure of the evaluation deduction $\mathcal{V} :: e \hookrightarrow v$ (see Figure 2.8 for the evaluation inference system). Each case of the induction constructs an object proof; since evaluation is closely related to proof reduction, this gives us a set of executable reductions for object proofs. Note that we are not proving a normalization theorem for the object logic, and because the semantics does not evaluate under functional abstractions, the soundness proof does not give even weak normal forms in the sense of Prawitz [Pra71].

Again we generalize to arbitrary contexts of extraction and prove the following lemma.

Lemma 3.16 *If* $\vdash e \hookrightarrow v$ *and* $\langle \Gamma, \Delta \rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ A\end{array}} \Downarrow e$ *then there is a proof* $\mathcal{P}'$ *such that* $\langle \Gamma, \Delta \rangle \vdash \boxed{\begin{array}{c}\mathcal{P}'\\ A\end{array}} \Downarrow v$

The following judgment represents the lemma in Elf:

```
tr_ev : eval M V -> extract (P:|- A) M -> extract (P':|- A) V -> type.
```

As for type soundness, we need a related lemma for extraction from individual terms. It is particularly simple because of the limited nature of our theory; in fact any expression extracted from an individual evaluates to itself.

Lemma 3.17 *If* $\Gamma \vdash t \Downarrow^i e$ *and* $e \hookrightarrow v$ *then* $\Gamma \vdash t \Downarrow^i v$

The Elf signature representing this lemma and its proof follows.

```
tr_ev_tm : eval M M' -> extract_tm T M -> extract_tm T M' -> type.

tr_ev_tm_z : tr_ev_tm ev_0 ex_zero ex_zero.
tr_ev_tm_s : tr_ev_tm (ev_s Ev) (ex_succ E) (ex_succ E') <- tr_ev_tm Ev E E'.
```

We do not have as clean an inversion principle as for type soundness: the form of an extracted program limits but does not uniquely determine the last inference of the extraction. Inspection of the extraction rules of Figures 2.16 – 2.19 gives the following correspondences.

Lemma 3.18 (Inversion for program extraction)

If $\mathcal{E} :: \langle \Gamma, \Delta \rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ A\end{array}} \Downarrow e$*, then the form of* e *and the form of* A *determine the last inference of* $\mathcal{E}$ *as follows.*

1. e *a variable :* ex-pvar
2. *If* $e = 0$ *or* $e = \mathbf{s}(e)'$ *then* e *cannot be extracted from an object proof.*
3. $e = \langle e_1, e_2 \rangle$ *: Inspection of the extraction rules gives* x∧I *or* x∃I*. But since the object proofs in question end in introduction rules, the outermost connective of* A *is enough to disambiguate.*
4. $e = \mathbf{fst}(e')$ *:* x∧E_L
5. $e = \mathbf{snd}(e')$ *:* x∧E_R
6. $e = \mathbf{spread}(e_1;\ x, y \,.\, e_2)$ *:* x∃E
7. $e = \mathbf{inl}(e')$ *:* x∨I_L
8. $e = \mathbf{inr}(e')$ *:* x∨I_R
9. $e = \mathbf{decide}(e_1;\ x \,.\, e_2;\ x \,.\, e_3)$ *:* x∨E

10. $e = \mathbf{lam}\, x \,.\, e'$: *The extraction must end in* x⊃I, x∀I, *or* x¬I. *But again these are all introduction rules, and the outermost connective of A uniquely determines the last inference.*

11. $e = \mathbf{nat\_ind}(e_1;\ x, y \,.\, e_2)$: xIND

12. $e = \mathbf{app}(e_1, e_2)$: x⊃E, x∀E, *or* x¬E

13. $e = ()$: x⊤

14. $e = \mathbf{any}(e)$: x⊥

15. $e = \mathbf{neg}$: xAXO

16. $e = \mathbf{axiom}$: x=R, x=Y, x=T, x=U, *or* xAXS

We also need a substitution lemma for program extractions. A fully detailed proof requires definitions of substitution for program expressions, logical formulas, natural deduction proofs, and extraction contexts. We omit the details as there is nothing unusual about them, except for contexts. Here we only define substitution for an individual variable in a context:

Definition 3.19 *(Substitution in contexts)*

$$[t/x](\Delta, \boxed{\begin{matrix}p\\A\end{matrix}} \Downarrow p') \equiv ([t/x]\Delta), \boxed{\begin{matrix}p\\ [t/x]A\end{matrix}} \Downarrow p'$$

Note also that substitution of a proof $\boxed{\begin{matrix}\mathcal{P}\\A\end{matrix}}$ for a proof variable $\boxed{\begin{matrix}p\\B\end{matrix}}$ is defined only if A and B are identical.

Lemma 3.20 (Substitution in program extractions)

i. If $\langle\, (\Gamma, x \Downarrow^i x'), \Delta \,\rangle \vdash \boxed{\begin{matrix}\mathcal{P}\\A\end{matrix}} \Downarrow e$ *and* $\Gamma \vdash t \Downarrow^i e'$

then $\langle\, \Gamma, [t/x]\Delta \,\rangle \vdash [t/x]\boxed{\begin{matrix}\mathcal{P}\\A\end{matrix}} \Downarrow [e'/x']e$

ii. If $\langle\, \Gamma, (\Delta, \boxed{\begin{matrix}q\\B\end{matrix}} \Downarrow q') \,\rangle \vdash \boxed{\begin{matrix}\mathcal{P}\\A\end{matrix}} \Downarrow e$ *and* $\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{matrix}\mathcal{Q}\\B\end{matrix}} \Downarrow e'$

then $\langle\, \Gamma, \Delta \,\rangle \vdash [\mathcal{Q}/q]\boxed{\begin{matrix}\mathcal{P}\\A\end{matrix}} \Downarrow [e'/q']e$

Corollary: *If x is not free in Δ,*

and $\langle\, (\Gamma, x \Downarrow^i x'), (\Delta, \boxed{\begin{matrix}q\\B\end{matrix}} \Downarrow q') \,\rangle \vdash \boxed{\begin{matrix}\mathcal{P}\\A\end{matrix}} \Downarrow e$ *and* $\Gamma \vdash t \Downarrow^i e_t$ *and* $\langle\, \Gamma, \Delta \,\rangle \vdash [t/x]\boxed{\begin{matrix}\mathcal{Q}\\B\end{matrix}} \Downarrow [e_t/x']e_Q$

then $\langle\, \Gamma, \Delta \,\rangle \vdash [t/x][\mathcal{Q}/q]\boxed{\begin{matrix}\mathcal{P}\\A\end{matrix}} \Downarrow [e_t/x'][e_Q/q']e$

The proof is again a structural induction on extraction deductions, and depends on a similar lemma guaranteeing that substitution of individual terms and proofs in natural deductions preserves validity.

The proof of the main lemma 3.16 is a straightforward induction on the structure of the evaluation deduction; we present only a sample of typical cases and their representation in Elf.

Base cases

In the case that $e \hookrightarrow e$ (**ev-lam**, **ev-pr**, **ev-unit**, **ev-ax**, and **ev-neg**) the conclusion is immediate. Each case is represented in Elf by a clause without subgoals:

```
tr_ev_lam : tr_ev ev_lam E E.
tr_ev_pr : tr_ev ev_pr E E.
tr_ev_unity : tr_ev ev_unity E E.
tr_ev_ax : tr_ev ev_ax E E.
tr_ev_neg : tr_ev ev_neg E E.
```

Congruence cases

Where evaluation merely descends through a non-binding constructor (**ev-pair**, **ev-inl**, and **ev-inr**) the conclusion follows easily by applying the induction hypothesis or Lemma 3.17 (evaluation soundness for individuals) to the subdeduction(s) of the evaluation. The cases **ev-inl** and **ev-inr** are trivial. We show the case where the evaluation ends in **ev-pair** in detail.

Assume $\mathcal{V} =$

$$\dfrac{\begin{matrix}\mathcal{V}_1 \\ e_1 \hookrightarrow v_1\end{matrix} \qquad \begin{matrix}\mathcal{V}_2 \\ e_2 \hookrightarrow v_2\end{matrix}}{\langle e_1, e_2 \rangle \hookrightarrow \langle v_1, v_2 \rangle}\,\textsf{ev-pair}$$

Then there are two subcases: the corresponding extraction $\mathcal{E}$ ends in x∧I or x∃I (Lemma 3.18).

Case $\mathcal{E} =$

$$\dfrac{\begin{matrix}\mathcal{E}_1 \\ \langle \Gamma, \Delta \rangle \vdash \boxed{\begin{matrix}\mathcal{P}_1 \\ A\end{matrix}} \Downarrow e_1\end{matrix} \qquad \begin{matrix}\mathcal{E}_2 \\ \langle \Gamma, \Delta \rangle \vdash \boxed{\begin{matrix}\mathcal{P}_2 \\ B\end{matrix}} \Downarrow e_2\end{matrix}}{\langle \Gamma, \Delta \rangle \vdash \boxed{\dfrac{\begin{matrix}\mathcal{P}_1 \\ A\end{matrix} \quad \begin{matrix}\mathcal{P}_2 \\ B\end{matrix}}{A \wedge B}\wedge\mathrm{I}} \Downarrow \langle e_1, e_2 \rangle}\,\mathrm{x}\wedge\mathrm{I}$$

By the induction hypothesis applied to $\mathcal{V}_1$ and $\mathcal{E}_1$ there is an object proof $\mathcal{P}_1'$ such that

$\mathcal{E}_1' :: \boxed{\begin{matrix}\mathcal{P}_1' \\ A\end{matrix}} \Downarrow v_1$. Similarly we have $\mathcal{E}_2' :: \boxed{\begin{matrix}\mathcal{P}_2' \\ A\end{matrix}} \Downarrow v_2$. Then construct

$$\cfrac{\begin{matrix}\mathcal{E}_1' \\ \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{matrix}\mathcal{P}_1' \\ A\end{matrix}} \Downarrow v_1\end{matrix} \qquad \begin{matrix}\mathcal{E}_2' \\ \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{matrix}\mathcal{P}_2' \\ B\end{matrix}} \Downarrow v_2\end{matrix}}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\cfrac{\begin{matrix}\mathcal{P}_1' & \mathcal{P}_2' \\ A & B\end{matrix}}{A \wedge B}\,{\scriptstyle \wedge\mathrm{I}}} \Downarrow \langle\, v_1, v_2 \,\rangle}\ \mathrm{x}\wedge\mathrm{I}$$

□

Case $\mathcal{E} =$

$$\cfrac{\begin{matrix}\mathcal{E}_1 \\ \Gamma \vdash t \Downarrow^i e_1\end{matrix} \qquad \begin{matrix}\mathcal{E}_2 \\ \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{matrix}\mathcal{P} \\ [t/x]A\end{matrix}} \Downarrow e_2\end{matrix}}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\cfrac{\begin{matrix}\mathcal{P} \\ [t/x]A\end{matrix}}{\exists x \,.\, A}\,{\scriptstyle \exists\mathrm{I}}} \Downarrow \langle\, e_1, e_2 \,\rangle}\ \mathrm{x}\exists\mathrm{I}$$

By Lemma 3.17 (evaluation soundness for individuals) we have $\mathcal{E}_1' :: \Gamma \vdash t \Downarrow^i v_1$. By the induction hypothesis for $\mathcal{V}_2$ and $\mathcal{E}_2$ there is an extraction $\mathcal{E}_2' :: \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{matrix}\mathcal{P}' \\ [t/x]A\end{matrix}} \Downarrow v_2$. Then construct

$$\cfrac{\begin{matrix}\mathcal{E}_1' \\ \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{matrix}\mathcal{P}_1' \\ A\end{matrix}} \Downarrow v_1\end{matrix} \qquad \begin{matrix}\mathcal{E}_2' \\ \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{matrix}\mathcal{P}_2' \\ B\end{matrix}} \Downarrow v_2\end{matrix}}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\cfrac{\begin{matrix}\mathcal{P}_1' & \mathcal{P}_2' \\ A & B\end{matrix}}{A \wedge B}\,{\scriptstyle \wedge\mathrm{I}}} \Downarrow \langle\, v_1, v_2 \,\rangle}\ \mathrm{x}\wedge\mathrm{I}$$

□

The constructive content of this case is represented by the following pair of Elf clauses.

```
tr_ev_pr_and : tr_ev (ev_pair Evr Evl) (ex_andi Er El) (ex_andi Er' El')
                <- tr_ev Evr Er Er'
                <- tr_ev Evl El El'.

tr_ev_pr_ex : tr_ev (ev_pair Evr Evl) (ex_existsi Er El) (ex_existsi Er' El')
               <- tr_ev_tm Evl El El'
               <- tr_ev Evr Er Er'.
```

The other congruence cases are similarly encoded in Elf.

```
tr_ev_inl : tr_ev (ev_inl Ev) (ex_oril E) (ex_oril E')
            <- tr_ev Ev E E'.

tr_ev_inr : tr_ev (ev_inr Ev) (ex_orir E) (ex_orir E')
            <- tr_ev Ev E E'.
```

The evaluation rule for successor (**ev-s**) is also a congruence, but the conclusion follows trivially since there is no rule that extracts an expression ($\mathbf{s}(e)$) from a proof.

Reduction cases

Proof reductions arise from eliminations (**ev-fst**, **ev-snd**, **ev-spread**, **ev-decide-l**, **ev-decide-r**, **ev-app-lam**, **ev-pr-z**, and **ev-pr-s**).

The cases for evaluation of $\mathbf{fst}(e)$ and $\mathbf{snd}(e)$ are dual; we show the reasoning for $\mathbf{fst}(e)$.

Case V =

$$\dfrac{\overset{\mathcal{V}_1}{e \hookrightarrow \langle\, v_1, v_2 \,\rangle}}{\mathbf{fst}(e) \hookrightarrow v_1}\,\textbf{ev-fst}$$

By inversion an extraction $\mathcal{E}$ of $\mathbf{fst}(e)$ has the form

$$\dfrac{\overset{\mathcal{E}_1}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ A \wedge B\end{array}} \Downarrow e}}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ \dfrac{A \wedge B}{A}\wedge\mathrm{E}_L\end{array}} \Downarrow \mathbf{fst}(e)}\,\mathrm{x}\wedge\mathrm{E}_L$$

By the induction hypothesis on $\mathcal{V}_1$ and $\mathcal{E}_1$ we have an extraction $\mathcal{E}'$ from a proof of $A \wedge B$ which by inversion has the following form:

$$\dfrac{\overset{\mathcal{E}'_1}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}_1\\ A\end{array}} \Downarrow v_1} \qquad \overset{\mathcal{E}'_2}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}_2\\ B\end{array}} \Downarrow v_2}}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}_1 \qquad \mathcal{P}_2\\ \dfrac{A \qquad B}{A \wedge B}\wedge\mathrm{I}\end{array}} \Downarrow \langle\, v_1, v_2 \,\rangle}\,\mathrm{x}\wedge\mathrm{I}$$

Then $\mathcal{P}_1$ is the required object proof with $\mathcal{E}'_1 :: \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}_1\\ A\end{array}} \Downarrow v_1$.

□

This case and its dual are represented in Elf by the clauses:

```
tr_ev_fst : tr_ev (ev_fst Ev) (ex_andel E) El
            <- tr_ev Ev E (ex_andi Er El).
tr_ev_snd : tr_ev (ev_snd Ev) (ex_ander E) Er
            <- tr_ev Ev E (ex_andi Er El).
```

Case $\mathcal{V} =$

$$\dfrac{\overset{\mathcal{V}_1}{e_1 \hookrightarrow \langle\, v_1, v_2 \,\rangle} \qquad \overset{\mathcal{V}_2}{[v_1/x][v_2/y]e_2 \hookrightarrow v}}{\mathbf{spread}(e_1;\ x, y\,.\,e_2) \hookrightarrow v}\ \textsf{ev-spread}$$

By inversion the extraction $\mathcal{E}$ has the form

$$\dfrac{\overset{\mathcal{E}_1}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{matrix}\mathcal{P}\\ \exists x'.A\end{matrix}} \Downarrow e_1} \qquad \overset{\mathcal{E}_2}{\langle\, (\Gamma, x' \Downarrow^i x), (\Delta, \boxed{\begin{matrix}y'\\ A\end{matrix}} \Downarrow y) \,\rangle \vdash \boxed{\begin{matrix}y'\\ A\\ \mathcal{Q}\\ C\end{matrix}} \Downarrow e_2}}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\dfrac{\begin{matrix}\mathcal{P}\\ \exists x'.A\end{matrix} \qquad \begin{matrix}\overline{\quad}\, y'\\ A\\ \mathcal{Q}\\ C\end{matrix}}{C}\ \exists\mathrm{E}^{y'}} \Downarrow \mathbf{spread}(e_1;\ x, y\,.\,e_2)}\ \textsf{x}\exists\textsf{E}$$

By the induction hypothesis on $\mathcal{V}_1$ and $\mathcal{E}_1$ we have an extraction of $\langle\, v_1, v_2 \,\rangle$ from a proof of $\exists x'.A$. By inversion the extraction has the following form.

$$\dfrac{\overset{\mathcal{E}_1'}{\Gamma \vdash t \Downarrow^i v_1} \qquad \overset{\mathcal{E}_2'}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{matrix}\mathcal{P}'\\ [t/x']A\end{matrix}} \Downarrow v_2}}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\dfrac{\begin{matrix}\mathcal{P}'\\ [t/x']A\end{matrix}}{\exists x'.A}\ \exists\mathrm{I}} \Downarrow \langle\, v_1, v_2 \,\rangle}\ \textsf{x}\exists\textsf{I}$$

Then Lemma 3.20 (substitution), applied to $\mathcal{E}_2$, $\mathcal{E}_1'$, and $\mathcal{E}_2'$ allows us to substitute t for x', v_1 for x, $\mathcal{P}'$ for y', and v_2 for y in $\mathcal{E}_2$. (Since x' is not free in Δ we can apply the corollary.) We obtain the extraction

$$\mathcal{E}' :: \langle\, \Gamma, \Delta \,\rangle \vdash [t/x'][\mathcal{P}'/y'] \boxed{\begin{matrix}y'\\ A\\ \mathcal{Q}\\ C\end{matrix}} \Downarrow [v_1/x][v_2/y]e_2$$

Since x' is not free in C, $[t/x'][\mathcal{P}'/y']\boxed{\begin{array}{c} y' \\ A \\ Q \\ C \end{array}}$ is a proof of C. Then applying the induction hypothesis to $\mathcal{V}_2$ and $\mathcal{E}'$ we have a Q' such that $\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c} Q' \\ C \end{array}} \Downarrow v$, as required.
□

The Elf representation of this case is an example of the use of term reconstruction to manage the details of the proof:

```
tr_ev_spread : tr_ev (ev_spread Ev_r Ev_p) (ex_existse E_ex E_min) E'
               <- tr_ev Ev_p E_ex (ex_existsi E Etm)
               <- tr_ev Ev_r (E_min _ _ Etm _ _ E) E'.
```

The terms that represent t, v_1, $\mathcal{P}'$, and v_2 of the informal proof need not be supplied (they correspond to the underscores in the input) since they are found during term and type reconstruction for the clause.

The two evaluation rules for **decide** are dual to one another. We show the case for left injection.

Case $\mathcal{V} =$

$$\cfrac{\begin{array}{c}\mathcal{V}_1 \\ e_1 \hookrightarrow \mathbf{inl}(v_1)\end{array} \qquad \begin{array}{c}\mathcal{V}_2 \\ [v_1/x]e_2 \hookrightarrow v\end{array}}{\mathbf{decide}(e_1;\ x\,.\,e_2;\ x\,.\,e_3) \hookrightarrow v}\ \textsf{ev-decide-l}$$

By inversion any extraction $\mathcal{E}$ of $\mathbf{decide}(e_1;\ x\,.\,e_2;\ x\,.\,e_3)$ has the form

$$\cfrac{\begin{array}{c}\mathcal{E}_1 \\ \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}_1 \\ A \vee B\end{array}} \Downarrow e_1\end{array} \quad \begin{array}{c}\mathcal{E}_2 \\ \langle\, \Gamma, (\Delta, \boxed{\begin{array}{c}p \\ A\end{array}} \Downarrow x) \,\rangle \vdash \boxed{\begin{array}{c}p \\ A \\ \mathcal{P}_2 \\ C\end{array}} \Downarrow e_2\end{array} \quad \begin{array}{c}\mathcal{E}_3 \\ \langle\, \Gamma, (\Delta, \boxed{\begin{array}{c}p \\ B\end{array}} \Downarrow x) \,\rangle \vdash \boxed{\begin{array}{c}p \\ B \\ \mathcal{P}_3 \\ C\end{array}} \Downarrow e_3\end{array}}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\cfrac{\begin{array}{c}\mathcal{P}_1 \\ A \vee B\end{array} \quad \begin{array}{c}\overset{-p}{A} \\ \mathcal{P}_2 \\ C\end{array} \quad \begin{array}{c}\overset{-p}{B} \\ \mathcal{P}_3 \\ C\end{array}}{C}\ {\vee}\mathrm{E}^p} \Downarrow \mathbf{decide}(e_1;\ x\,.\,e_2;\ x\,.\,e_3)}\ \mathsf{x}{\vee}\mathsf{E}$$

The induction hypothesis for $\mathcal{V}_1$ and $\mathcal{E}_1$, together with inversion, gives $\mathcal{E}_1' =$

$$\cfrac{\begin{array}{c}\mathcal{E}_1'' \\ \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P} \\ A\end{array}} \Downarrow v_1\end{array}}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\cfrac{\begin{array}{c}\mathcal{P}_1' \\ A\end{array}}{A \vee B}\ {\vee}\mathrm{I}_L} \Downarrow \mathbf{inl}(v_1)}\ \mathsf{x}{\vee}\mathsf{I}_L$$

We use the Substitution Lemma applied to $\mathcal{E}_2$ and $\mathcal{E}_1''$ to obtain

$$\mathcal{E}_2' :: \langle\, \Gamma, \Delta \,\rangle \vdash [\mathcal{P}/p]\boxed{\begin{matrix} p \\ A \\ \mathcal{P}_2 \\ C \end{matrix}} \Downarrow [v_1/x]e_2$$

Then the induction hypothesis applied to $\mathcal{V}_2$ and $\mathcal{E}_2'$ yields an extraction of v from a proof of C as required.
□

This case and its dual are represented in Elf as follows. Again term reconstruction relieves us of the need to supply all the terms involved in the use of the Substitution Lemma.

```
tr_ev_decl : tr_ev (ev_dec_l Ev Ev_l) (ex_ore Er El E) E''
              <- tr_ev Ev_l E (ex_oril E')
              <- tr_ev Ev (El _ _ E') E''.
tr_ev_decr : tr_ev (ev_dec_r Ev Ev_r) (ex_ore Er El E) E''
              <- tr_ev Ev_r E (ex_orir E')
              <- tr_ev Ev (Er _ _ E') E''.
```

Case $\mathcal{V} =$

$$\dfrac{\begin{matrix}\mathcal{V}_1 \\ e_1 \hookrightarrow \mathbf{lam}\, x \,.\, e\end{matrix} \qquad \begin{matrix}\mathcal{V}_2 \\ e_2 \hookrightarrow v_2\end{matrix} \qquad \begin{matrix}\mathcal{V}_3 \\ [v_2/x]e \hookrightarrow v\end{matrix}}{\mathbf{app}(e_1, e_2) \hookrightarrow v}\ \textsf{ev-app-lam}$$

By inversion there are three possible forms for the corresponding extraction $\mathcal{E}$: the object proof can end in $\supset$E, $\neg$E, or $\forall$E. The proofs for these cases present no new features; in each subcase we employ inversion and substitution to obtain an extraction deduction to which the induction hypothesis applies, and the resulting object proof is the obvious reduction of the original one.
□

The Elf code that implements the three subcases is:

```
tr_ev_lam_all : tr_ev (ev_app_lam Ev_r Ev_arg Ev_lam) (ex_foralle Etm E) E''
                 <- tr_ev Ev_lam E (ex_foralli E')
                 <- tr_ev_tm Ev_arg Etm Etm'
                 <- tr_ev Ev_r (E' _ _ Etm') E''.

tr_ev_lam_imp : tr_ev (ev_app_lam Ev_r Ev_arg Ev_lam) (ex_impliese E_min E_maj) E'
                 <- tr_ev Ev_lam E_maj (ex_impliesi E)
                 <- tr_ev Ev_arg E_min E_min'
                 <- tr_ev Ev_r (E _ _ E_min') E'.

tr_ev_lam_not : tr_ev (ev_app_lam Ev_r Ev_arg Ev_lam) (ex_note E_pos E_neg) E'
```

```
<- tr_ev Ev_lam E_neg (ex_noti E)
<- tr_ev Ev_arg E_pos E_pos'
<- tr_ev Ev_r (E _ _ E_pos') E'.
```

Case $\mathcal{V} =$

$$\dfrac{\begin{array}{c}\mathcal{V}_1\\ e_1 \hookrightarrow \mathbf{nat\_ind}(e_z;\ x,y\,.\,e_s)\end{array} \quad \begin{array}{c}\mathcal{V}_2\\ e_2 \hookrightarrow \mathbf{s}(v_2)\end{array} \quad \begin{array}{c}\mathcal{V}_3\\ \mathbf{app}((\mathbf{nat\_ind}(e_z;\ x,y\,.\,e_s)), v_2) \hookrightarrow v_3\end{array} \quad \begin{array}{c}\mathcal{V}_4\\ [v_2/x][v_3/y]e_s \hookrightarrow v\end{array}}{\mathbf{app}(e_1, e_2) \hookrightarrow v}\,\textsf{ev-pr-s}$$

By inversion there are three possible forms for the corresponding extraction $\mathcal{E}$: the object proof $\mathcal{P}$ can end in $\supset$E, $\neg$E, or $\forall$E.

If $\mathcal{P}$ ends in $\supset$E, then e_1 is extracted from a proof of an implication $A \supset B$. Since $e_1 \hookrightarrow$ $\mathbf{nat\_ind}(e_z;\ x,y\,.\,e_s)$, by induction there is another proof of $A \supset B$ from which we can extract $\mathbf{nat\_ind}(e_z;\ x,y\,.\,e_s)$. By inversion this extraction must end in xIND; but this is impossible. The same reasoning applies to $\neg$E; therefore the only case to consider is $\forall$E. Here we see that the inversion principle is too weak, and we must appeal to the induction hypothesis to prove the totality of the translation function we are constructing. This is a weakness in the proof that could be avoided if we used a different encoding of induction in the object logic.

$\mathcal{E} =$

$$\dfrac{\begin{array}{c}\mathcal{E}_1\\ \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ \forall x'\,.\,A\end{array}} \Downarrow e_1\end{array} \qquad \begin{array}{c}\mathcal{E}_2\\ \Gamma \vdash t \Downarrow^i e_2\end{array}}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}\\ \dfrac{\forall x'\,.\,A}{[t/x']A}\,\forall\text{E}\end{array}} \Downarrow \mathbf{app}(e_1, e_2)}\,\textsf{x}\forall\textsf{E}$$

By the induction hypothesis applied to $\mathcal{V}_1$ and $\mathcal{E}_1$ we have an extraction $\mathcal{E}_1'$ which by inversion has the form:
$\mathcal{E}_1' =$

$$\dfrac{\begin{array}{c}\mathcal{E}_b\\ \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}_1\\ [\text{zero}/x']A\end{array}} \Downarrow e_z\end{array} \qquad \begin{array}{c}\mathcal{E}_s\\ \langle\, (\Gamma, x' \Downarrow^i x), (\Delta, \boxed{\begin{array}{c}y'\\ A\end{array}} \Downarrow y) \,\rangle \vdash \boxed{\begin{array}{c}y'\\ A\\ \mathcal{P}_s\\ [\text{succ}\ x'/x']A\end{array}} \Downarrow e_s\end{array}}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\dfrac{\begin{array}{c}\mathcal{P}_1\\ [\text{zero}/x']A\end{array} \qquad \begin{array}{c}\overline{A}^{\,y'}\\ \mathcal{P}_s\\ [\text{succ}\ x'/x']A\end{array}}{\forall x'\,.\,A}\,\text{IND}^{y'}} \Downarrow \mathbf{nat\_ind}(e_z;\ x,y\,.\,e_s)}\,\textsf{xIND}$$

By lemma 3.17 applied to $\mathcal{V}_2$ and $\mathcal{E}_2$ we have an extraction of $\mathbf{s}(v_2)$. This must have the form

$$\dfrac{\begin{array}{c}\mathcal{E}_2' \\ \Gamma \vdash t' \Downarrow^i v_2\end{array}}{\Gamma \vdash \text{succ } t' \Downarrow^i \mathbf{s}(v_2)}\,\textsf{xtm-s}$$

where $t = \text{succ } t'$.

From $\mathcal{E}_1'$ and $\mathcal{E}_2'$ we construct the extraction
$\mathcal{E}_3 =$

$$\dfrac{\begin{array}{c}\mathcal{E}_1' \\ \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P} \\ \forall x' .\, A\end{array}} \Downarrow \mathbf{nat\_ind}(e_z;\ x, y \,.\, e_s)\end{array} \qquad \begin{array}{c}\mathcal{E}_2' \\ \Gamma \vdash t' \Downarrow^i v_2\end{array}}{\langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P} \\ \dfrac{\forall x' .\, A}{[t'/x']A}\,\forall\text{E}\end{array}} \Downarrow \mathbf{app}(\mathbf{nat\_ind}(e_z;\ x, y \,.\, e_s), v_2)}\,\textsf{x}\forall\textsf{E}$$

Then we can apply the induction hypothesis to $\mathcal{V}_3$ and $\mathcal{E}_3$ to obtain

$$\mathcal{E}_3' :: \langle\, \Gamma, \Delta \,\rangle \vdash \boxed{\begin{array}{c}\mathcal{P}' \\ \dfrac{\forall x' .\, A}{[t'/x']A}\,\forall\text{E}\end{array}} \Downarrow v_3$$

By the substitution lemma applied to $\mathcal{E}_s$ with $\mathcal{E}_2'$ and $\mathcal{E}_3'$ we have

$$\mathcal{E}_s' :: \langle\, \Gamma, \Delta \,\rangle \vdash [t'/x'][\mathcal{P}'/y'] \boxed{\begin{array}{c}y' \\ A \\ \mathcal{P}_s \\ [\text{succ } x'/x']A\end{array}} \Downarrow [v_2/x][v_3/y]e_s$$

Since $\text{succ } t' = t$ the conclusion of the object proof is $[t/x]A$. Then the induction hypothesis applied to $\mathcal{V}_4$ and $\mathcal{E}_s'$ gives an extraction of v from a proof of $[t/x]A$ as required.
□

The following Elf code represents the constructive content of this case:

```
tr_ev_prs :
 tr_ev (ev_pr_s Ev_s Ev Ev_p) (ex_foralle (ex_succ Etm) E) E'
 <- tr_ev Ev_p E (ex_ind Es Ez)
 <- tr_ev_tm Ev (ex_succ Etm) (ex_succ Etm')
 <- tr_ev Ev_s (Es _ _ Etm' _ _ (ex_foralle Etm' (ex_ind Es Ez))) E'.
```

3.4 Discussion

We have shown a fragment of the metatheory of the implementation. Type soundness is illustrated by the following diagram. The colon ":" is LF typing; **pf** Φ is the type of object proofs with conclusion Φ. The dotted arrow corresponds to the typing derivation constructed by the soundness proof.

$$\begin{tikzcd}
\text{(Proof) } \mathcal{P} \arrow[r, ":"] \arrow[d, "\Downarrow"'] & \text{(LF type) } \mathsf{pf}\,\Phi \arrow[d, "\Downarrow^t"] \\
\text{(Program) } M \arrow[r, dashed, "\in"'] & \text{(Type) } \tau
\end{tikzcd}$$

The following diagram illustrates evaluation soundness in the same way. The dotted arrow labelled "reduce" corresponds to the relation between the given proof $\mathcal{P}$ and the proof $\mathcal{P}'$ constructed by the soundness proof.

$$\begin{tikzcd}
& \text{(LF type) } \mathsf{pf}\,\Phi & \\
\text{(Proof) } \mathcal{P} \arrow[ur, ":"] \arrow[rr, dashed, "\text{reduce}"'] \arrow[d, "\Downarrow"'] & & \text{(Proof) } \mathcal{P}' \arrow[ul, ":"'] \arrow[d, dashed, "\Downarrow"] \\
\text{(Program) } M \arrow[rr, "\hookrightarrow"'] & & \text{(Program) } V
\end{tikzcd}$$

An immediate consequence of type and evaluation soundness together is subject reduction for the set of programming language expressions that can be extracted from proofs.

Other correctness theorems could be encoded in the same way. (For an implementation in the same style of the metatheory of an encoding of Mini-ML similar to our programming language encoding, see [PR92].) Type and evaluation soundness would then allow us to carry the metatheory over from the domain of programs to the domain of proofs; for instance, we could get a very weak normal form for natural deduction proofs from a proof that evaluation of an extracted program produces a "value" (for some appropriate definition of value).

Chapter 4

Proof transformation

The proof transformations considered in this thesis are for the most part syntactic transformations expressible by means of higher-order patterns in the spirit of Huet and Lang [HL78]. The next chapter explores the application of these transformations in program development without giving details of their implementation. This chapter gives a detailed account of one proof transformation and its implementation in Elf. Section 4.1 informally describes the transformation and shows how to formalize a simple version of it in Elf and apply it to a proof. This rather naive formulation needs improvements described in subsequent sections. Section 4.2 describes how to avoid generating detours in the result proof that require the application of expensive proof normalizations. Section 4.3 discusses modifications to the encoding for the suppression of unwanted computation in the extracted program.

4.1 Tail recursion introduction

A well-known programming strategy for the improvement of a recursive function definition is the introduction of an accumulator argument to achieve a tail-recursive form, which can be compiled to a loop. This strategy has long been studied by researchers in transformational programming, for example in [BD77], [Bir84], and many others. There is a transformation that performs the analogous optimization at the level of proofs; its application to a small example program shows that in some cases the proof transformation can perform more optimization than techniques restricted to the program level.

4.1.1 Language extensions

In order to work out the example it is necessary to add a small theory of lists of natural numbers to the logic, and to extend the theory of natural numbers. These theories are somewhat *ad hoc*, and are selected for convenience in working out this particular example. This is a result of the very limited capabilities of the base system, which does not support inductive definitions, unlike fully developed systems like Nuprl and Coq. The extended logic is a many-sorted first-order logic with sorts {nats, lists}, where lists are monomorphic. This is obtained by modifying and extending the abstract syntax of the core logic (page 17). The modification is simple: the individual variables x

become sort-annotated variables x^i, and the equality symbol and quantifiers are also sort-annotated $(=_i, \forall_i, \exists_i)$. The logical syntax is then extended as follows:

Individuals	$t ::=$	$\ldots \mid \text{pred}\ t \mid t_1 - t_2 \mid \text{signum}\ t \mid t_1 \geq t_2$
Lists	$l ::=$	$x^\ell \mid [] \mid t :: l \mid \text{if}\ t\ \text{then}\ l_1\ \text{else}\ l_2$
Propositions	$A ::=$	$\ldots \mid \forall_\ell x^\ell . A \mid \exists_\ell x^\ell . A \mid l_1 =_\ell l_2 \mid t \in l$

The predecessor function is denoted by "pred". The signum operator is used to build conditional functions; signum $x = 0$ if and only if $x = 0$ and signum $x = 1$ if and only if $x \neq 0$. This is used to axiomatize $\geq$ as a function on natural numbers, which in conjunction with the list-valued arithmetic switch (if-then-else) permits the construction of the conditional function needed for the example of this chapter. The syntax for lists is ML-like, with [] denoting the empty list and :: denoting the cons operator.

Figures 4.1 and 4.2 show the inference rules for the sorts of natural numbers and lists used in the formalization of the proofs for the example. Inference rules for equality and quantifiers over lists are omitted since they are identical to those for natural numbers except for the sort. List induction is parametric in l^ℓ and x^i; thus neither parameter may occur free in any open assumption on which $[\ []/l^\ell]A$ or $[x^i :: l^\ell/l^\ell]A$ depends. For the remainder of the chapter we omit sort annotations whenever the context makes the intended sort clear.

The programming language is extended accordingly. Extraction of programs from proofs is not difficult to extend to the new language. There are now three primitive relations – equalities for natural numbers and lists, and the list membership relation; these are all considered uninformative for the purposes of extraction.

e	$::=$	$\ldots \mid [] \mid e_1 :: e_2 \mid$	*Lists*
		$\mathbf{list\_ind}(e_1;\ x, y, z . e_2) \mid$	*Primitive recursion on lists*
		$e_1 - e_2 \mid$	*Subtraction*
		$e_1 \geq e_2 \mid$	*Comparison*
		$\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$	*Arithmetic switch*

For the sake of readability the programs of this chapter are shown in ML syntax. A straightforward mechanical translation can be defined, but we also freely rename variables, introduce let-bindings, and use pattern-matching. Natural numbers are represented as ML integers, and lists as ML lists. Pairs are represented as ML tuples; $\mathbf{fst}(e)$ and $\mathbf{snd}(e)$ correspond to ML `#1 e` and `#2 e`. A primitive recursion over natural numbers $\mathbf{nat\_ind}(e_1;\ x, y . e_2)$ is represented as

```
fun f 0 = e1
  | f n = let val y = (f (n-1)) in e2 end
```

where `f` is a new variable, and e_1 is represented as `e1` and e_2 as `e2`. Primitive recursion over lists is translated analogously. The union type $\tau_1|\tau_2$ with its injection constructors **inl** and **inr** is represented by the following datatype definition:

```
datatype ('a, 'b) union = inl of 'a | inr of 'b;
```

$$\frac{t_1 =_i t_2 \qquad A}{[t_2/t_1]A}\ {=}\mathrm{S}$$

$$\frac{}{\text{pred zero} =_i \text{zero}}\ \mathrm{PredZ}$$

$$\frac{}{\text{pred succ } t =_i t}\ \mathrm{PredS}$$

$$\frac{}{t - \text{zero} =_i t}\ \mathrm{SubZ}$$

$$\frac{}{t_1 - \text{succ } t_2 =_i \text{pred } (t_1 - t_2)}\ \mathrm{SubS}$$

$$\frac{}{\text{signum } t =_i (\text{succ zero}) - ((\text{succ zero}) - t)}\ \mathrm{Signum}$$

$$\frac{}{t_1 \geq t_2 =_i \text{signum } (t_2 - t_1)}\ \mathrm{GEq}$$

Figure 4.1: Extension to theory of natural numbers

$$\dfrac{[\,[\,]/l^{\ell}]A \qquad \begin{array}{c}\overline{A}^{\,p}\\ \vdots\\ [x^{i} :: l^{\ell}/l^{\ell}]A\end{array}}{\forall_{\ell} l^{\ell} .\, A}\,\text{LIND}^{p}$$

$$\dfrac{}{\neg t \in [\,]}\,\text{AXnil}$$

$$\dfrac{}{t \in t :: l}\,\text{AXhd}$$

$$\dfrac{t_1 \in l}{t_1 \in t_2 :: l}\,\text{AXtl}$$

$$\dfrac{t_1 \in t_2 :: l \qquad [t_2/t_1]A \qquad \begin{array}{c}\overline{x^{i} \in l}^{\,p}\\ \vdots\\ [x^{i}/t_1]A\end{array}}{A}\,\in\text{E}$$

$$\dfrac{}{(\text{if zero then } l_1 \text{ else } l_2) = l_1}\,\text{LSz}$$

$$\dfrac{}{(\text{if (succ zero) then } l_1 \text{ else } l_2) = l_2}\,\text{LSs}$$

Figure 4.2: Fragment of a theory of lists of natural numbers

Elf encodings of these language extensions and the extracted programs of this chapter are given in Appendix D and Appendix E.

4.1.2 A simple proof and program

The presentation of the tail recursion proof transformation is organized around a small example: a program to select all elements of a list that satisfy some constraint. For concreteness and simplicity, it is expedient to fix the selection criterion for the example program, and develop a function that, given a list l of natural numbers, computes a list of all elements of l that are at least 2. Following is the specification with a simple proof.

Specification 4.1

$$\forall l \,.\, \exists r \,.\, [\forall y \,.\, y \in r \Leftrightarrow (y \in l \ \wedge \ y \geq 2)]$$

Proof 4.2 The proof is by induction on the list l. If l is empty, choose r to be empty as well. Otherwise $l = x :: l'$ and, by the induction hypothesis, there is an r' such that $\forall y \,.\, y \in r' \Leftrightarrow (y \in l' \ \wedge \ y \geq 2)$. Then if $x \geq 2$ let r be $x :: r'$; otherwise let r be r'. □

The following program is extracted from a formalized version of the proof:

Program 4.3

```
fun select [] = []
  | select (x::l) =
    let val r = (select l) in
     if x >= 2 then x::r else r
    end
```

This can easily be transformed at the program level to tail-recursive form using the fold-unfold system [BD77]. One introduces an auxiliary function with an extra "accumulator" argument that builds up the result as control is passed down the chain of recursive calls. Here is a sketch of the fold-unfold method (`@` is the ML list append operator).

First define an auxiliary function with accumulator argument `k`.

```
fun sel l k = k@(select l)
```

Unfold the definition of `select` in the body of `sel`:

```
fun sel [] k = k@[]
  | sel (x::l) k = k@(if x >= 2 then x::(select l)
                      else (select l))
```

Then equational reasoning yields:

```
fun sel [] k = k
  | sel (x::l) k = if x >= 2 then (k@[x])@(select l)
                   else k@(select l)
```

Folding yields a new recursive definition of `sel`; the original function `select` can now be defined in terms of it:

Program 4.4

```
fun sel [] k = k
  | sel (x::l) k = if x >= 2 then (sel l (k@[x]))
                   else (sel l k)

fun select l = sel l []
```

The transformed definition of `sel` is tail-recursive, but is inefficient since it uses the append function. There is a similar development that shifts the inefficiency to the base case of the recursion, giving:

```
fun sel [] k = (reverse k)
  | sel (x::l) k = if x >= 2 then (sel l (x::k))
                   else (sel l k)
```

where `reverse` is defined in the obvious way.

The specification requires only that the result contain the elements of l that are at least 2, so `sel` could equally well be defined to accumulate its result via the recursive call `sel l' (x::k)`, without reversing the result list. But since this change does not preserve functional equivalence, it is not straightforward to accomplish via purely syntactic program transformation. One could avoid the difficulty by observing that l is treated here as a set, not a list, and working in a more appropriate algebra. Depending on the context in which the function is used, this may or may not be easy to do. The use of a proof transformation yields a similar tail-recursive program that computes with lists but does not use the append or reverse functions.

4.1.3 Informal description

How might one transform Proof 4.2 to obtain a proof whose realizing program is tail-recursive? In a constructive proof of $\forall l \,.\, \exists r \,.\, \Phi(l,r)$ by list induction, the inductive step must construct some term t such that for arbitrary x, $\Phi(x :: l, t)$, using the induction hypothesis $\exists r' \,.\, \Phi(l, r')$ as a premise. The term t corresponds to the value that is returned by the extracted program. A recursive call in the program is extracted from a use of the induction hypothesis in the proof. A program is tail-recursive when it does no computation with the results of its recursive calls. Correspondingly, an inductive proof is realized by a tail-recursive program when the term t is identical to the parameter r' of the induction hypothesis. That is, if the proof appeals to the induction hypothesis to obtain a parameter r' satisfying the specification for l, and shows that *the same value* r' also satisfies the specification for $x :: l$ for any x, the extracted function is tail-recursive. An induction proof whose extracted program is tail-recursive is *tail-inductive.*

In Appendix C we give an Elf program to recognize tail-recursive object programs, and discuss the partial representation in Elf of a proof that the transformation of this chapter yields a tail-recursive program if it succeeds.

For example, Proof 4.2 appeals to the induction hypothesis with l' for l to obtain r' such that $\forall y . y \in r' \Leftrightarrow (y \in l' \ \wedge \ y \geq 2)$; this corresponds to the recursive call `select l` in the extracted program. The program is not tail recursive because in the case that $x \geq 2$, r' does not satisfy the specification for $x :: l'$ and the proof constructs the term $x :: r'$. This construction corresponds to the computation `(x::r)` (where `r` is bound to the result of the recursive call) in the program.

To find a tail-inductive proof, it is necessary to modify the induction hypothesis. That is, a new specification is needed that implies the original one. The proof of the new specification is an inductive subproof that corresponds to the auxiliary function definition `sel` of the fold-unfold derivation. However, much of the new inductive subproof consists of inferences from the original induction proof: it is built by a form of *lemma insertion* [Pfe90].

Lemma insertion is the process of inserting new inferences (proofs of lemmas) into a given proof in such a way that the new proof is valid if the original proofs and the new inferences are valid. Perhaps the simplest example of such a transformation is the following: given a proof $\mathcal{I}_1$ of $\Phi \supset \Psi$ and a proof $\mathcal{I}_2$ of $\Psi \supset \Phi$, we can transform

$$\begin{array}{c}\mathcal{D}\\ \vdots\\ \Phi\end{array}$$

to

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \vdots\\ \Phi\end{array} \qquad \begin{array}{c}\mathcal{I}_1\\ \vdots\\ \Phi \supset \Psi\end{array}}{\Psi} \qquad \begin{array}{c}\mathcal{I}_2\\ \vdots\\ \Psi \supset \Phi\end{array}}{\Phi}$$

This is an example of a proof transformation that does not change the specification Φ, but only the method of proof, which determines the form of the implementation. The correctness of the transformation is a trivial meta-theorem for any useful formalization of the proof system. The transformed proof has a different computational content than the original one. It still contains all the original computational content (since $\mathcal{D}$ is a subproof); typically proof reduction is applied after lemma insertion in order to remove redundant computational content and obtain a program with different properties. Thus lemma insertion is analogous to the definition of an auxiliary function followed by unfolding in a program derivation in the fold-unfold system [BD77]. It creates a context that is exploited by subsequent transformations.

Lemma insertion and proof reduction by themselves determine nothing about program development – it is the choice of lemmas and of proof methods that express programming knowledge. The formalization/implementation of tail recursion introduction presented here does not capture all of the programming knowledge involved: it introduces an improved recursion structure if the user can provide the right lemmas and proofs for insertion. By expressing some minimum requirements for the form of these lemmas and proofs it can provide some guidance to a user, reducing the search space for a solution.

Thus before the lemma insertion transformation can be applied, some insight and knowledge of the problem domain must be applied to invent the right lemmas and their proofs.

First, it is necessary to find a function h that accumulates the desired result of the computation in an auxiliary argument k as computation descends through the recursive call sequence. Suppose the original proof concludes $\forall l\,.\,\exists r\,.\,\Phi(l,r)$ where l and r are lists. Then h should take the result k accumulated so far for l and combine it with the head x of the current list so that $\Phi(l,k) \supset \Phi(l \circ [x], h(x,k))$(where $\circ$ stands for the list append operation). Somewhat unexpectedly, a proof of this implication is not required for the proof transformation; it is only a guide to the choice of h. This choice can often be "read off" from the original proof. Since the original proof is not tail-inductive, there is a term $f(x,r)$ that plays a crucial role in the inductive step. The proof appeals to the induction hypothesis to obtain a parameter r such that $\Phi(l,r)$, and then constructs $f(x,r)$ such that for all x $\Phi(x :: l, f(x,r))$. If $\Phi(l,r)$ does not depend on the order of elements in l and r then $h = f$ is an acceptable choice.

The second preparatory step is to choose a formula $\Psi(k,r,s)$ that expresses the fact that k is an accumulator argument and s is the new final result. The transformed proof will prove

$$\forall l\,.\,\forall k\,.\,\exists s\,.\,\exists r\,.\,\Phi(l,r) \;\wedge\; \Psi(k,r,s)$$

by an induction obtained by lemma insertion from the original induction. From this will follow $\forall l\,.\,\exists r\,.\,\Phi(l,r)$, but not by the obvious trivial method of eliminations. This part of the proof will show that for an appropriate initial term k_0, $\Phi(l,s)$ follows from $\Phi(l,r) \;\wedge\; \Psi(k_0,r,s)$; thus the tail-recursive computation of s can replace the non-tail-recursive computation of r. This is stated more precisely below.

A simple choice for $\Psi(k,r,s)$ is $s = r + k$ where "+" is chosen from an algebra appropriate to the problem domain. This can often be done – perhaps always, if one is willing to shift the problem specification to another algebra – but it is not required; equality is often more than is needed. Ψ need only express a relation for which three crucial lemmas hold. Informally these are:

1. There is a value k_0 such that if $\Phi(l,r) \;\wedge\; \Psi(k_0,r,s)$ then $\Phi(l,s)$, for any r, s, and l. This ensures that the new computation satisfies the old specification.

2. If $\Psi(h(x,k),r,s)$ then $\Psi(k,f(x,r),s)$. This is the crucial requirement for achieving tail recursion. It ensures that in an induction step for $\forall l\,.\,\forall k\,.\,\exists s\,.\,\exists r\,.\,\Phi(l,r) \;\wedge\; \Psi(k,r,s)$ a parameter s given by the induction hypothesis for l also satisfies the conclusion for $x :: l$.

3. There is a function $s_0(k)$ such that $\Psi(k, r_0, s_0(k))$ for all k, where r_0 is the term that witnesses the existence proof for the base case of the original proof. In all our examples $s_0(k) = k$.

If the `select` example were posed in terms of sets rather than lists, a good choice of $\Psi(k,r,s)$ would be $(s = r \cup k)$. In that case the proof would proceed by structural induction on a type of finite sets with constructor $\uplus$ (so $r \uplus \{x\}$ denotes the disjoint union of r and $\{x\}$). In this formulation $f(x,r) \equiv r \uplus \{x\}$; choosing $h(x,k) \equiv \{x\} \uplus k$ makes it trivial to show the crucial step $s = (r \uplus \{x\}) \cup k$ from $s = r \cup (\{x\} \uplus k)$ using the associativity of set union. Whenever Ψ can be expressed as an equality $s = F(k,r)$ the crucial requirement 2 above becomes $F(h(x,k),r) = F(k,f(x,r))$. A comparison with Bird's analysis [Bir84] of general accumulation strategies in program transformation shows that this requirement is a special case of a "continuity condition" for the accumulation strategy; when this case holds, the strategy results in a tail-recursive program. Thus if Ψ can be expressed as an equality, pure program transformation techniques have the same power as the proof transformation described here.

Since the example is posed in terms of lists instead of sets, it is not possible to use equality and for this example proof transformation accomplishes more than syntactic program transformation.

Once the lemmas are formulated and proved, the rest of the transformation is mechanical, and can be implemented in Elf as a rewrite rule. The rest of this section is a description of the rule in terms of proof schemas, with its application to the `select` example. The following section gives its formalization and implementation in Elf.

The original induction proof has the following form:

Proof schema 4.5

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_{\mathbf{B}}\\ \vdots\\ \boldsymbol{\Phi}([],\mathbf{r}_0)\end{array}}{\exists r\,.\,\boldsymbol{\Phi}([],r)}\,\exists\mathrm{I} \qquad \dfrac{\dfrac{}{\exists r\,.\,\boldsymbol{\Phi}(l,r)}\,p \qquad \dfrac{\begin{array}{c}\dfrac{}{\boldsymbol{\Phi}(l,r)}\,q\\ \vdots\\ \mathcal{D}_{\mathbf{S}}\\ \vdots\\ \boldsymbol{\Phi}(x::l,\mathbf{f}(x,r))\end{array}}{\exists r\,.\,\boldsymbol{\Phi}(x::l,r)}\,\exists\mathrm{I}}{\exists r\,.\,\boldsymbol{\Phi}(x::l,r)}\,\exists\mathrm{E}^q}{\forall l\,.\,\exists r\,.\,\boldsymbol{\Phi}(l,r)}\,\mathrm{LIND}^p$$

In this schema boldface symbols are (possibly higher-order) pattern variables. Where a pattern variable is applied to some terms $t_1 \dots t_i$ as in $\boldsymbol{\Phi}(l,r)$, the whole application is a pattern in which the terms $t_1 \dots t_i$ may occur free. The font used for the pattern variable indicates whether the pattern may match a formula (upper-case Greek letters), an individual (Roman letters, e.g., $\mathbf{f}(x,r)$), or a part of a proof (script style capitals, e.g., $\mathcal{D}$) in the object logic. A schema of the form

$$\begin{array}{c}\mathcal{D}\\ \vdots\\ \boldsymbol{\Psi}\end{array}$$

matches an object proof ending in a formula $\boldsymbol{\Psi}$; any variables free in $\boldsymbol{\Psi}$ may also occur free in $\mathcal{D}$. Similarly a schema of the form

$$\begin{array}{c}\boldsymbol{\Psi}\\ \vdots\\ \mathcal{D}\\ \vdots\\ \boldsymbol{\Psi}'\end{array}$$

matches a proof fragment that depends on Ψ and has Ψ' as conclusion; any variables free in Ψ or Ψ' may occur free in $\mathcal{D}$. Ordinary italic letters stand for bound variables and parameters of the object logic.

For the `select` example, $\mathbf{\Phi}(l,r)$ matches $\forall y\,.\,y \in r \Leftrightarrow (y \in l \;\wedge\; y \geq 2)$. Note that higher-order patterns respect the binding properties of the object language. Thus in $\mathbf{\Phi}(t_1,t_2) = \forall y\,.\,y \in t_2 \Leftrightarrow (y \in t_1 \;\wedge\; y \geq 2)$, y may not occur free in t_1 or t_2. In the Elf implementation the use of function objects in the LF calculus to represent higher-order patterns, in combination with higher-order abstract syntax, enforces this restriction. When the meaning is clear from context, we will write "the term $\mathbf{t}$" to mean a term that matches the pattern $\mathbf{t}$, and similarly for formulas and proofs.

The next three schemas describe the lemmas concerned with the auxiliary specification Ψ, which expresses the relation between the original result r, the accumulator argument k, and the new result s.

1. To recover the original specification from the new conclusion of the induction the transformation requires a proof $\mathcal{I}_\mathbf{R}$ of

   Lemma schema 4.6 (Recovery lemma)

   $$\forall l\,.\forall r\,.\forall s\,.\,(\mathbf{\Phi}(l,r) \;\wedge\; \mathbf{\Psi}(\mathbf{k_0},r,s)) \supset \mathbf{\Phi}(l,s)$$

   for some term $\mathbf{k_0}$.

   For the `select` example $\mathbf{\Psi}(k,r,s)$ is $\forall y\,.\,y \in s \Leftrightarrow (y \in r \;\vee\; y \in k)$; thus the transformed induction sub-proof will prove:

   Specification 4.7

   $$\forall l\,.\forall k\,.\exists s\,.\exists r\,.[\forall y\,.\,y \in r \Leftrightarrow (y \in l \;\wedge\; y \geq 2)] \;\wedge\; [\forall y\,.\,y \in s \Leftrightarrow (y \in r \;\vee\; y \in k)]$$

   The term $\mathbf{k_0}$ is the empty list [], and it is easy to show

   $$\forall l\,.\forall r\,.\forall s\,.[(\forall y\,.\,y \in r \Leftrightarrow (y \in l \;\wedge\; y \geq 2)) \;\wedge\; (\forall y\,.\,y \in s \Leftrightarrow (y \in r \;\vee\; y \in []))] \supset \\ \forall y\,.\,y \in s \Leftrightarrow (y \in l \;\wedge\; y \geq 2)$$

   Operationally this proof corresponds to a redefinition of the original function in terms of an auxiliary function which is extracted from the induction subproof. It determines an initial value $\mathbf{k_0}$ for the accumulator argument k.

2. The second obligation imposed by the transformation is a proof $\mathcal{I}_\mathbf{S}$ of

   Lemma schema 4.8 (Inductive case lemma)

   $$\forall x\,.\forall k\,.\forall r\,.\forall s\,.\,\mathbf{\Psi}(\mathbf{h}(x,k),r,s) \supset \mathbf{\Psi}(k,\mathbf{f}(x,r),s)$$

   where the pattern $\mathbf{f}(x,r)$ *is given by the original proof matching Schema 4.5 and the pattern* $\mathbf{h}(x,k)$ *is supplied by the user.*

For the **select** example, $\mathbf{f}(x,r) = \textbf{if } x \geq 2 \textbf{ then } x :: r \textbf{ else } r$. Choosing $\mathbf{h}(x,k) = \textbf{if } x \geq 2 \textbf{ then } x :: k \textbf{ else } k$ leads to the following lemma, which is easily proved:

$$\begin{array}{l}\forall x\,.\,\forall k\,.\,\forall r\,.\,\forall s\,.\,(\forall y\,.\,y \in s \Leftrightarrow (y \in r \ \vee \ y \in (\textbf{if } x \geq 2 \textbf{ then } x :: k \textbf{ else } k))) \supset \\ \qquad\qquad (\forall y\,.\,y \in s \Leftrightarrow (y \in (\textbf{if } x \geq 2 \textbf{ then } x :: r \textbf{ else } r) \ \vee \ y \in k))\end{array}$$

Viewed operationally, $\mathbf{h}$ determines the value bound to the accumulator argument k in the recursive call of the extracted program.

3. For the base case of the new inductive subproof, the transformation requires a proof $\mathcal{I}_{\mathbf{B}}$ of

   Lemma schema 4.9 (Base case lemma)

   $$\forall k\,.\,\Psi(k, \mathbf{r_0}, \mathbf{s_0}(k))$$

   $\mathbf{r_0}$ is the witness term for the base case of the original proof matching Schema 4.5; $\mathbf{s_0}$ is supplied by the user.

   Operationally, $\mathbf{s_0}(k)$ is the value that will be returned in the base case of the recursion.

   For the list selection example, $\mathbf{r_0}$ is the empty list [], since that is the witness for the base case in Proof 4.2. No further processing is needed of the value accumulated in k, so we choose the identity function for lists as the value for $\mathbf{s_0}$. Then, applying Specification 4.7, we have $\Psi(k, [], k) = \forall y\,.\,y \in k \Leftrightarrow (y \in [] \ \vee \ y \in k)$, which is trivial to prove for all k.

The proof transformation inserts the proof $\mathcal{I}_{\mathbf{B}}$ into the base case subproof of the original proof, the proof $\mathcal{I}_{\mathbf{S}}$ into the step case subproof, and constructs a proof of the original specification from the new specification using the proof $\mathcal{I}_{\mathbf{R}}$. The result has the following form:

Proof schema 4.10

$$\begin{array}{c}
\dfrac{\begin{array}{c}\mathcal{D}_{\mathbf{B'}}\\ \vdots \\ \forall k\,.\,\exists s\,.\,\exists r\,.\,\Phi([],r) \ \wedge \ \Psi(k,r,s)\end{array}
\qquad
\begin{array}{c}
\overline{\overline{\hphantom{\Phi(l,r) \ \wedge \ \Psi(\mathbf{h}(x,k),r,s)}}}\ 1 \\
\Phi(l,r) \ \wedge \ \Psi(\mathbf{h}(x,k),r,s) \\ \vdots \\ \mathcal{D}_{\mathbf{S'}} \\ \vdots \\
\Phi(x :: l, \mathbf{f}(x,r)) \ \wedge \ \Psi(k, \mathbf{f}(x,r), s) \\
\hline\hline
\forall k\,.\,\exists s\,.\,\exists r\,.\,\Phi(x :: l, r) \ \wedge \ \Psi(k,r,s)
\end{array}}{\forall l\,.\,\forall k\,.\,\exists s\,.\,\exists r\,.\,\Phi(l,r) \ \wedge \ \Psi(k,r,s)}\ \text{LIND}^1 \\
\vdots \\
\mathcal{D}_{\mathbf{R}} \\
\vdots \\
\forall l\,.\,\exists r\,.\,\Phi(l,r)
\end{array}$$

$\mathcal{D}_{\mathbf{B}'}$ is constructed from $\mathcal{D}_{\mathbf{B}}$ of Proof schema 4.5 and $\mathcal{I}_{\mathbf{B}}$; $\mathcal{D}_{\mathbf{S}'}$ from $\mathcal{D}_{\mathbf{S}}$ and $\mathcal{I}_{\mathbf{S}}$, and $\mathcal{D}_{\mathbf{R}}$ from $\mathcal{I}_{\mathbf{R}}$ in the obvious way. We use double horizontal lines in the schema to elide trivial quantifier elimination and introduction steps.

The following is an informal statement of the new proof of Specification 4.1 obtained by applying this transformation to the `select` example:

Proof 4.11 We first show

$$\forall l\,.\,\forall k\,.\,\exists s\,.\,\exists r\,.\,[\forall y\,.\,y \in r \Leftrightarrow (y \in l \ \wedge\ y \geq 2)] \ \wedge\ [\forall y\,.\,y \in s \Leftrightarrow (y \in r \ \vee\ y \in k)]$$

by induction on l. If $l = []$ then choose $s = k$ and $r = []$. Otherwise $l = x :: l'$. If $x \geq 2$, then by instantiating the induction hypothesis with l' and $x :: k$, we obtain a list r' containing all elements ≥ 2 of l' and a list s' containing all the elements of r' and $x :: k$. Then choose $r = x :: r'$ and $s = s'$. Otherwise $\neg x \geq 2$; again by the induction hypothesis there is a list r' as before, and a list s' containing all the elements of r' and k. Let $r = r'$ and $s = s'$.

Now for any list l, there are lists s, r such that

$$[\forall y\,.\,y \in r \Leftrightarrow (y \in l \ \wedge\ y \geq 2)] \ \wedge\ [\forall y\,.\,y \in s \Leftrightarrow (y \in r \ \vee\ y \in [])]$$

Since s contains exactly the elements of r, we have

$$\forall y\,.\,y \in s \Leftrightarrow (y \in l \ \wedge\ y \geq 2)$$

and

$$\forall l\,.\,\exists r\,.\,[\forall y\,.\,y \in r \Leftrightarrow (y \in l \ \wedge\ y \geq 2)]$$

□

The form of the inductive reasoning in this proof determines a tail-recursive computation, but only for the parameter s: the term s' given by the induction hypothesis for l', $x :: k$ satisfies the specification for $x :: l'$, k, but the term r' does not; we must construct $x :: r'$ as witness for the inner existential quantifier. To look at it another way, we have developed a program that computes the answer we want twice: once by the original, non-tail-recursive method, and once by a new method that (on its own) is tail-recursive. Thus it may seem that the transformation has not accomplished anything! However, there is a proof transformation that removes the unwanted computation associated with the inner existential quantifier. This transformation is quite general and does not impose any further proof obligations on the user. It introduces a double negation to produce an inductive proof of the following:

$$\forall l\,.\,\forall k\,.\,\exists s\,.\,\neg\neg\exists r\,.\,[\forall y\,.\,y \in r \Leftrightarrow (y \in l \ \wedge\ y \geq 2)] \ \wedge\ [\forall y\,.\,y \in s \Leftrightarrow (y \in r \ \vee\ y \in k)]$$

Section 4.3 discusses this transformation in detail.

The rest of this chapter describes the implementation of the transformation. Applied to the example the implementation yields the following extracted program:

Program 4.12

```
fun sel l =
 let fun sel' nil = (fn k => k)
       | sel' (x::l') = let val p = (sel' l') in
           fn k => (p (if x >= 2 then (x::k) else k)) end
 in (sel' l nil)
 end;
```

Like the result of the fold-unfold derivation (Program 4.4), this program is tail-recursive. But it is more efficient because the partial result is accumulated by the application of the list constructor instead of the append function. The difference is a consequence of working with the specification and its proof throughout the development rather than restricting the reasoning to the properties of the program alone. It is an instance of the phenomenon noted by Goad [Goa80]: using proof transformation he demonstrated that a program can be specialized to a particular class of input values by exploiting the fact that there is no need to preserve functional equivalence of the successive stages of program development.

4.1.4 Formalization

The first step in the formalization of the transformation is to declare a judgment that relates the input proofs and individual terms to the output proof. By giving the types of the terms related by the judgment the declaration provides static (type-checking time) correctness guarantees. (For the Elf syntax of lists and list quantification see Figure 4.3.)

```
tail_rec :
 {R0} {S0} {K0} {F} {H}
 {Phi: ilist -> ilist -> o}
 {Psi: ilist -> ilist -> ilist -> o}
 %% Input proof:
 |- (lforall [l] lexists [r] (Phi l r))
 %% Base case lemma:
 -> |- (lforall [k] Psi k R0 (S0 k))
 %% Inductive case lemma:
 -> |- (forall [x] lforall [k] lforall [r] lforall [s]
        implies (Psi (H x k) r s) (Psi k (F x r) s))
 %% Recovery lemma:
 -> |- (lforall [l] lforall [r] lforall [s]
     implies (and (Phi l r) (Psi K0 r s)) (Phi l s))
 %% Output proof:
 -> |- (lforall [l] lexists [r] (Phi l r))
 -> type.
```

The logic variables in the declaration correspond to the pattern variables of the proof and lemma schemas of the previous section; e.g., R0 corresponds to the pattern variable r_0, Phi corresponds

```
ilist : type.                     %lists of natural numbers

nl : ilist.                       %empty list
cns : i -> ilist -> ilist.        %cons

leq : ilist -> ilist -> o.        %equality for lists

lforall : (ilist -> o) -> o.      %universal quantifier
lexists : (ilist -> o) -> o.      %existential quantifier

lforalli : ({x:ilist} |- (A x)) -> |- (lforall A).
lforalle : {T:ilist} |- (lforall A) -> |- (A T).

lexistsi : {A:ilist -> o} {T:ilist} |- (A T) -> |- (lexists A).
lexistse : ({x:ilist} |- (A x) -> |- C) -> |- (lexists A) -> |- C.

lind :    {A:ilist -> o}
          |- (A nl)
       -> ({l:ilist} |- (A l) -> {x:i} |- (A (cns x l)))
       -> |- (lforall A).
```

Figure 4.3: Elf encoding of lists of natural numbers

to Φ, and so on. The types of the individuals and functions on them `R0` ... `H` need not be given since the Elf type reconstruction algorithm can infer them from their contexts in the declaration.

The declaration ensures that the judgment relates (a term representing) a proof of $\forall l \,.\, \exists r \,.\, \Phi(l,r)$ (which can be thought of as an input), other input terms representing individuals, functions on individuals, and the proofs of the lemmas to be inserted, and another proof (which can be thought of as an output) of $\forall l \,.\, \exists r \,.\, \Phi(l,r)$. Assuming that the logic is correctly encoded, if a query of the form "`?- tail_rec ... <proof-term> ... P`" succeeds, the LF type system ensures that the variable `P` is bound to a term that represents a valid proof of the same proposition.

The other step in the encoding is to write a clause describing the forms of the terms related by the judgment `tail_rec`. This can be done by transcribing the schemas of the previous section into Elf. We then have a one-clause Elf "program" that implements the transformation as a rewrite rule.

First we encode Proof schema 4.5, which describes the input proof.

```
tl_rec : tail_rec
 ...
 %% Input proof:
 (lind ([l] lexists [r] Phi l r)
       (lexistsi ([r] Phi nl r) R0 Db)
       ([l] [p: |- (lexists [r] Phi l r)] [x]
        (lexistse ([r] [q: |- (Phi l r)]
         lexistsi _ (F x r) (Ds x l r q))
         p)))
 ...
```

For the sake of readability this declaration includes some type and term information that can be inferred by Elf. Again the upper-case letters are Elf logic variables, and correspond to the pattern variables of Proof schema 4.5. Thus for instance `Ds` corresponds to the subproof $\mathcal{D}_{\mathrm{S}}$. Since the transformation is encoded as a pattern match with no subgoals, these variables are instantiated only by higher-order unification with the object proof term supplied in the query. Thus `Phi` is bound to a function from two lists to the matrix of the Π^2_0 formula proved, `R0` to the list that witnesses the base case of the induction, `Db` to the proof of the base case, and so on.

Some care must be taken when coding transformations this way to avoid encountering unification problems that are not solved by the Elf implementation. If an encoding restricts its use of logic variables for pattern matching to *higher-order patterns* (in the sense of [Pfe91b]), only deterministic unification problems will arise during execution. These problems fall within a decidable subcase of higher-order unification (which is undecidable in general), discovered by Miller [Mil91] for the simply typed lambda calculus and extended to LF by Pfenning. The main idea of the restriction is that variables subject to instantiation during search (like `Ds`) should occur only as *generalized variables*. Roughly speaking, a generalized variable is a term of the form $xy_1 \ldots y_n$ where x is subject to instantiation and the y_i are distinct bound variables (not subject to instantiation). In the example above, the subterm `Phi l r` is a generalized variable, while `Phi nl r` is not. The latter term could lead to nondeterminism if unified with one containing a logic variable; but this does not cause problems when the clause is used as intended, that is, the input proof and lemmas are given as ground terms in the query.

Note that the subproof $\mathcal{D}_S$ is represented by a logic variable of functional type in order to express the fact that the parameters x, l, r, and q may appear free in this part of the proof.

The encodings for the proofs of the three lemmas to be inserted are trivial: a logic variable for each proof is all that is needed. This is because the transformation does not depend on the structure of these proofs. We use Elf logic variables `I_b`, `I_s`, and `I_r` to correspond to the pattern variables $\mathcal{I}_B$, $\mathcal{I}_S$, and $\mathcal{I}_R$ respectively. Similarly the transformation does not depend on the structure of the individual terms and the functions on them ($\mathbf{r}_0$ and so on), so these too are encoded as logic variables. The clause at this stage reads:

```
tl_rec : tail_rec
 R0 S0 K0 F H
 Phi Psi
 %% Input proof:
 (lind ([l] lexists [r] Phi l r)
       %% Base case:
       (lexistsi ([r] Phi nl r) R0 Db)
       %% Step case:
       ([l] [p: |- (lexists [r] Phi l r)] [x]
        (lexistse ([r] [q: |- (Phi l r)]
         lexistsi _ (F x r) (Ds x l r q))
         p)))
 %% Base case lemma:
 I_b
 %% Step case lemma:
 I_s
 %% Recovery lemma:
 I_r
 ...
```

Once all the inputs are described and named, the pattern of Proof schema 4.10, which describes the output proof, can be formalized and added as the final argument of the clause `tl_rec`. Where the schema elides the details of how the lemmas are inserted, the formalization must give them explicitly.

Consider the base case of the output proof. It is assembled from the subproof $\mathcal{D}_B$ of the input proof and the auxiliary proof $\mathcal{I}_B$. It must prove

$$\forall k\,.\,\exists s\,.\,\exists r\,.\,\Phi([\,], r) \;\wedge\; \Psi(k, r, s)$$

Since $\mathcal{I}_B$ proves $\forall k\,.\,\Psi(k, \mathbf{r}_0, \mathbf{s}_0(k))$ it is necessary to eliminate the quantifier before using it. $\mathcal{D}_B$, which proves $\Phi([\,], \mathbf{r}_0)$ can be used directly. Thus the whole base case of the new induction is described by the pattern:

```
(lforalli [k] lexistsi _ (S0 k) (lexistsi _ R0 (andi Db  (lforalle k I_b))))
```

Similarly before using $\mathcal{I}_{\mathbf{S}}$, which proves

$$\forall x\,.\,\forall k\,.\,\forall r\,.\,\forall s\,.\,\Psi(\mathbf{h}(x,k),r,s) \supset \Psi(k,\mathbf{f}(x,r),s)$$

the transformation must perform eliminations on the induction hypothesis

$$\forall k\,.\,\exists s\,.\,\exists r\,.\,\Phi(l,r) \ \wedge\ \Psi(k,r,s)$$

to obtain $\Psi(\mathbf{h}(x,k),r,s)$. The implication derived from $\mathcal{I}_{\mathbf{S}}$ can then be used to obtain $\Psi(k,\mathbf{f}(x,r),s)$. The subproof also needs $\Phi(x :: l,\mathbf{f}(x,r))$, which is obtained by again performing eliminations on the induction hypothesis and applying $\mathcal{D}_{\mathbf{S}}$. Finally the appropriate quantifiers must be introduced.

Then the step case of the induction is encoded as:

```
[l] % list parameter
[p] % induction hypothesis
[x] % element parameter
(lforalli [k]
 (lexistse ([s] [q']
  (lexistse ([r] [q]
   lexistsi _ s
    (lexistsi _ (F x r)
      (andi
       (Ds x l r (andel q))
       (impliese
        (lforalle s (lforalle r (lforalle k (foralle x I_s))))
        (ander q)))))
   q'))
 (lforalle (H x k) p)))
```

For similar reasons in order to use the proof $\mathcal{I}_{\mathbf{R}}$ of

$$\forall l\,.\,\forall r\,.\,\forall s\,.\,(\Phi(l,r) \ \wedge\ \Psi(\mathbf{k_0},r,s)) \supset \Phi(l,s)$$

to obtain a proof of the specification $\forall l\,.\,\exists r\,.\,\Phi(l,r)$ it is necessary to eliminate the quantifiers before the implication can be used.

The whole clause defining the rewrite rule in Elf is shown in Figure 4.4. With this clause and formalizations *Psi*, I_B, I_S, I_R of Ψ, $\mathcal{I}_{\mathbf{B}}$, $\mathcal{I}_{\mathbf{S}}$, and $\mathcal{I}_{\mathbf{R}}$ respectively, we can transform a formalization P of Proof 4.2 to a formalization of Proof 4.11 by submitting the following query to Elf:

`?- tail_rec R0 S0 F H K0 Phi` *Psi* P I_B I_S I_R `Q.`

Here the italic variables (e.g., P) stand for closed LF object terms provided by the user. The text in typewriter font (e.g., `R0`) is input as-is by the user; thus the variables `R0`...`Phi` and `Q` are Elf logic variables. If the query succeeds the Elf interpreter displays the terms bound to them in the course of the search. Note that when the specification *Psi*, the input proof P and the auxiliary proofs I_B, I_S, and I_R are given as ground terms in the query, it is not necessary to provide terms for `R0`, `S0`, `F`, `H`, `K0`, or `Phi`; they are found by unification in the course of term and type reconstruction.

```
tl_rec :
 tail_rec
 R0 S0 K0 F H
 Phi Psi
 %% Input proof:
 (lind ([l] lexists [r] Phi l r)
       %% Base case:
       (lexistsi ([r] Phi nl r) R0 Db)
       %% Step case:
       ([l] [p: |- (lexists [r] Phi l r)] [x]
        (lexistse ([r] [q: |- (Phi l r)]
         lexistsi _ (F x r) (Ds x l r q))
         p)))
 %% Lemma proofs:
 I_b I_s I_r
 %% Output proof:
 (lforalli [l]
  %% Recovery of original specification:
  lexistse ([s] [p']
   lexistse ([r] [p]
    (lexistsi _ s (impliese (lforalle s (lforalle r (lforalle l I_r))) p)))
    p')
   (lforalle K0
     (lforalle l
      %% New inductive sub-proof:
      (lind ([l] lforall ([k] lexists [s] lexists [r] and (Phi l r) (Psi k r s)))
        %% Base case:
        (lforalli [k] lexistsi _ (S0 k) (lexistsi _ R0 (andi Db  (lforalle k I_b))))
        %% Step case:
        ([l] [p] [x]
         (lforalli [k]
          (lexistse ([s] [q']
           (lexistse ([r] [q]
            lexistsi _ s
             (lexistsi _ (F x r)
               (andi
                (Ds x l r (andel q))
                (impliese
                 (lforalle s (lforalle r (lforalle k (foralle x I_s))))
                 (ander q)))))
            q'))
          (lforalle (H x k) p)))))))).
```

Figure 4.4: Elf implementation of the tail recursion transformation

On the other hand, it is necessary to give the predicate Ψ explicitly because the corresponding logic variable `Psi` does not occur as a generalized variable in the declaration of the judgment `tail_rec`.

The proof term bound to `Q` has some undesirable features. It is likely to contain "detours": each inserted lemma will typically be a maximum formula in the sense of Prawitz [Pra71], being the conclusion of an introduction rule and the premise of the corresponding elimination rule. These detours are reflected in the extracted program as redices like the following fragment, extracted from the transformed list selection example proof. The reader is not intended to decipher to computational meaning of this; what is important is the appearance of the unreduced application of the function `u`. This reflects the quantifier and implication eliminations immediately following introductions in the source proof.

```
...
let fun u x' z1 z2 z3 q y =
 case (q y) of (inl p') => inl ()
  | (inr p') =>
    case (if x' >= 2 then inl () else inr ())
      of (inl q') => if y = x' then inl () else inr ()
       | (inr q') => inr ()
in
  (s, (if x >= 2 then x::r else r, (u x k r s p2)))
end
...
```

The next section discusses how to avoid introducing these detours.

The more serious defect is that, as we mentioned before, because `Q` contains a proof of the informative formula

$$\forall l\,.\,\forall k\,.\,\exists s\,.\,\exists r\,.\,\Phi(l,r)\ \wedge\ \Psi(k,r,s)$$

the program extracted from `Q` must compute both witnesses r and s, doing all the computation of the original program as well as the new computation introduced by the transformation. Section 4.3 describes how to eliminate the computation of r by introducing double negations into parts of the inductive subproof.

4.2 Suppressing detours in the output proof

Transformations on proofs often introduce detours which can be removed by normalization. But since normalization is expensive, and may remove redices which the user would prefer to keep, it is better to encode a transformation so that it does not introduce unnecessary detours in the result proof.

In the case of the tail-recursion transformation, examining the pattern for the output proof shows that detours in the form of quantifier or implication introductions immediately followed by eliminations are likely to be introduced when the lemma proofs `I_b`, `I_s`, and `I_r` are used. These detours can be avoided by lifting the universal quantifiers of the object logic to the meta-level. That is, the judgment takes *schematic* proofs of the lemmas, represented as functions of dependent type from list terms to proofs. In addition, for Lemma schemas 4.6 and 4.8, we lift the object-logic implication to the meta-level. Then the beta reductions performed by Elf in the course of producing canonical terms take the place of explicit proof normalization. The lifted representation is faithful to the object logic; this is a consequence of the fact that constructive minimal many-sorted predicate logic is representable in LF via the propositions-as-types interpretation, as described in, e.g., [Bar91]. In this interpretation universal quantification in the object logic is represented by Π-quantification and implication by simple function types.

We describe lifting for the simplest case first. Instead of requiring a proof of Lemma schema 4.9:

$$\forall k\,.\,\Psi(k,\mathbf{r_0},\mathbf{s_0}(k))$$

we require a function from a term representing a list to a proof of the lemma, with type:

$$\Pi k\mathsf{:ilist}\,.\,\mathsf{pf}\ \Psi(k,\mathbf{r_0},\mathbf{s_0}(k))$$

In Elf's concrete syntax this is:

```
{k:ilist} |- (Psi k R0 (S0 k))
```

With this representation, the pattern for the output proof no longer needs the $\forall$E rule. Instead the schematic proof, represented as an LF function, is applied to a term to produce an object-level proof. Thus the subterm in the original encoding

```
(lforalle k I_b)
```

becomes

```
(I_b k)
```

Similarly, the type of the proof of Lemma schema 4.8 becomes:

```
{x} {k} {r} {s} |- (Psi (H x k) r s) -> |- (Psi k (F x r) s)
```

Instead of the sequence of eliminations

```
(impliese (lforalle s (lforalle r (lforalle k (foralle x I_s)))) (ander q))
```

the pattern for the output proof contains the application

```
(I_s x k r s (ander q))
```

```
tl_rec : tail_rec
  R0 S0 K0 F H
  Phi Psi
 %% Input proof:
 (lind ([l] lexists ([r] Phi l r))
       %% Base case:
       (lexistsi ([r] Phi nl r) R0 Db)
       %% Step case:
       ([l] [p: |- (lexists ([r] Phi l r))] [x]
        (lexistse ([r] [q: |- (Phi l r)]
         lexistsi _ (F x r) (Ds x l r q))
         p)))
 %% Lemma proofs:
 I_b I_s I_r
 %% Output proof:
 (lforalli ([l]
  %% Recovery of original specification:
  lexistse ([s] [p']
   lexistse ([r] [p]
    (lexistsi _ s (I_r l r s p)))
    p')
   (lforalle K0
     (lforalle l
      %% New inductive sub-proof:
      (lind ([l] lforall ([k] lexists ([s] lexists ([r] and (Phi l r) (Psi k r s)))))
        %% Base case:
        (lforalli ([k] lexistsi _ (S0 k) (lexistsi _ R0 (andi Db (I_b k)))))
        %% Step case:
        ([l] [p] [x]
         (lforalli [k]
          (lexistse ([s] [q']
           (lexistse ([r] [q]
            lexistsi _ s
             (lexistsi _ (F x r)
               (andi (Ds x l r (andel q)) (I_s x k r s (ander q)))))
            q'))
          (lforalle (H x k) p)))))))))).
```

Figure 4.5: Lifted Elf implementation of tail-recursion

```
lift_all : |- (forall A) -> ({x} |- (A x)) -> type.

lift_all_down : lift_all (foralli P) P.
lift_all_up :   lift_all P ([x] foralle x P).
```

Figure 4.6: Correctness of lifting

The encoding of Lemma schema 4.6 is treated in the same way. The complete declaration of the lifted form of the judgment `tail_rec` is shown in Figure 4.5.

The soundness of this lifting technique is easy to see: from any schematic proof $\Pi x\,.\,\Phi$ a proof of $\forall x\,.\,\Phi$ can be built by the application of the constructor $\forall$I, and a similar argument holds for implication. Completeness follows from the observation that for any proof P of $\forall x\,.\,\Phi$ there is a schematic proof $\lambda x\,.(\forall\text{E}\ x\ P)$ of type $\Pi x\,.\,\Phi$, and similarly for implication. Correctness of lifting can be formulated and proved very simply using the techniques of Chapter 3. We show the fragment for the case of universal quantification over terms of sort `i` in Figure 4.6. The clause `lift_all_down` shows soundness since it gives an object proof for an arbitrary lifted proof `P`, and `lift_all_up` shows completeness, giving a lifted proof for any object proof. The clauses defining `lift_all` could be executed as proof transformations to lift arbitrary given lemmas, but in the special case (but in practice a very likely case) of a lemma's proof terminating in $\forall$I, the clause `lift_all_up` would introduce a detour in the resulting proof. In this case the proof `P` given by `lift_all_down` is the desired lifted version.

Pragmatically, the benefits of lifting are twofold: it improves efficiency by reducing the size of the proof terms to be processed during transformation and program extraction, and results in simpler, more compact and intelligible encodings of transformations. The same technique has been used by Pfenning [Pfe92a] in an Elf implementation of Plotkin's continuation-passing style conversion [Plo75], following an analysis by Danvy and Filinski [DF92].

Program 4.13

```
fun sel l =
 let fun sel' nil = (fn k => (k, (nil, fn x => inr ())))
       | sel' (x :: l') = let val p = (sel' l') in
           fn k =>
            let val (s, p1) = (p (if x >= 2 then (x::k) else k)) in
            let val (r, p2) = p1 in
            let val r' = (if x >= 2 then (x::r) else r) in
            let val p3 =
             fn y => case (p2 y) of
                     (inl p') => (inl ())
                   | (inr p') =>
                     case (if x >= 2 then inl () else inr ())
                       of (inl q') => (if y = x then inl () else inr ())
                        | (inr q') => inr () in
             (s, (r', p3)) end end end end end
 in let val (s, p) = (sel' l nil) in
     let val (r, p1) = p in s
 end end end;
```

Figure 4.7: Superfluous computation in **select** example

4.3 Suppressing unwanted computation

As we noted above, the transformed proof still contains all the structure of the original, including the now superfluous inner existence proof for r. As a result the extracted program is still not tail-recursive. This can be seen in Figure 4.7, which shows the program extracted from the proof for the select example. Roughly speaking, the outlined code corresponds to the existence proof for r. More precisely, the auxiliary function `sel'` has type:

```
int list -> int list -> int list * (int list * (int -> (unit,unit) union))
```

But all we really need is `int list -> int list -> int list`; the extra structure is generated by the existence proof for r. (Removing uninformative terms during extraction does not help here because the presence of an existential quantifier and a disjunction in the specification makes it "informative".) To avoid the unwanted computation, we further transform the inductive subproof to obtain a proof of

$$\forall l . \forall k . \exists s . \neg\neg\exists r . \Phi(l, r) \wedge \Psi(k, r, s)$$

Introducing the double negation of the existential quantifier for r prevents the extraction of a program to compute r, since we have defined negative formulas as uninformative.

The double negation introduction can be done easily since the following are derived rules of the object logic:

$$\dfrac{A}{\neg\neg A}\,\neg\neg\mathrm{I} \qquad\qquad \dfrac{\begin{array}{c}\overline{A}^{\,p}\\ \vdots\\ B\end{array} \qquad \neg\neg A}{\neg\neg B}\,\neg\neg\mathrm{I}^{p}_{\mathrm{hyp}}$$

Their derivations give the needed proof transformations in the form of rewrite rules. These are applied to the appropriate subproofs – those that have $\exists r \,.\, \Phi(t, r)$ for some term t – as the output proof is constructed.

First, given a closed proof $\mathcal{D}$ with end-formula A, transform $\mathcal{D}$ to:

Proof schema 4.14

$$\dfrac{\dfrac{\overline{\neg A}^{\,p} \qquad \begin{array}{c}\mathcal{D}\\ A\end{array}}{\bot}\,\neg\mathrm{E}}{\neg\neg A}\,\neg\mathrm{I}^{p}$$

Similarly, given an open proof depending on the premise A:

$$\begin{array}{c}A\\ \vdots\\ \mathcal{D}\\ \vdots\\ B\end{array}$$

transform it to:

Proof schema 4.15

$$\dfrac{\dfrac{\neg\neg A \qquad \dfrac{\dfrac{\overline{\neg B}^{\,q} \qquad \begin{array}{c}\overline{A}^{\,p}\\ \vdots\\ \mathcal{D}\\ \vdots\\ B\end{array}}{\bot}\,\neg\mathrm{E}}{\neg A}\,\neg\mathrm{I}^{p}}{\bot}\,\neg\mathrm{E}}{\neg\neg B}\,\neg\mathrm{I}^{q}$$

Translation into Elf is straightforward:

```
doubleneg : |- A -> |- (not (not A)) -> type.

dbl_neg : doubleneg D (noti ([p] (note p D))).

doubleneg_1_prem :
 (|- A -> |- B) -> (|- (not (not A)) -> |- (not (not B))) -> type.

dbl_neg_1_p :
 doubleneg_1_prem D ([pnnA] (noti ([pnB] note pnnA (noti ([pA] note pnB (D pA)))))).
```

We use these double negation transformations in tail-recursion introduction to hide the unwanted computation of the witness r. We change the transformation so that it produces an inductive subproof of

$$\forall l\,.\,\forall k\,.\,\exists s\,.\,\neg\neg\exists r\,.\,\Phi(l,r)\ \wedge\ \Psi(k,r,s)$$

To obtain this proof we apply the double negation transformation doubleneg to the part of the base case of the induction that proves the existence of r. Without the use of double negation, the base case has the following form:

Proof schema 4.16

$$\genfrac{}{}{2pt}{0}{\dfrac{\dfrac{\begin{matrix}\mathcal{D}_{\mathbf{B}} \\ \vdots \\ \Phi([],\mathbf{r_0})\end{matrix} \qquad \begin{matrix}\mathcal{I}_{\mathbf{B}} \\ \vdots \\ \Psi(k,\mathbf{r_0},\mathbf{s_0}(k))\end{matrix}}{\Phi([],\mathbf{r_0})\ \wedge\ \Psi(k,\mathbf{r_0},\mathbf{s_0}(k))}\,{\scriptstyle \wedge\mathrm{I}}}{\exists r\,.\,\Phi([],r)\ \wedge\ \Psi(k,r,\mathbf{s_0}(k))}\,{\scriptstyle \exists\mathrm{I}}}{\forall k\,.\,\exists s\,.\,\exists r\,.\,\Phi([],r)\ \wedge\ \Psi(k,r,s)}$$

(The double horizontal bar elides the obvious quantifier introductions.)

With double negation it has the form:

Proof schema 4.17

$$\genfrac{}{}{2pt}{0}{\dfrac{\dfrac{\dfrac{}{\neg\exists r\,.\,\Phi([],r)\ \wedge\ \Psi(k,r,\mathbf{s_0}(k))}\,{\scriptstyle p} \qquad \dfrac{\dfrac{\begin{matrix}\mathcal{D}_{\mathbf{B}} \\ \vdots \\ \Phi([],\mathbf{r_0})\end{matrix} \qquad \begin{matrix}\mathcal{I}_{\mathbf{B}} \\ \vdots \\ \Psi(k,\mathbf{r_0},\mathbf{s_0}(k))\end{matrix}}{\Phi([],\mathbf{r_0})\ \wedge\ \Psi(k,\mathbf{r_0},\mathbf{s_0}(k))}\,{\scriptstyle \wedge\mathrm{I}}}{\exists r\,.\,\Phi([],r)\ \wedge\ \Psi(k,r,\mathbf{s_0}(k))}\,{\scriptstyle \exists\mathrm{I}}}{\perp}\,{\scriptstyle \neg\mathrm{E}}}{\neg\neg\exists r\,.\,\Phi([],r)\ \wedge\ \Psi(k,r,\mathbf{s_0}(k))}\,{\scriptstyle \neg\mathrm{I}^{p}}}{\forall k\,.\,\exists s\,.\,\neg\neg\exists r\,.\,\Phi([],r)\ \wedge\ \Psi(k,r,s)}$$

In a similar way we apply the transformation **doubleneg_1_prem** to the portion of the step case that proves the existence of r.

Since the recovery lemma $\mathcal{I}_{\mathbf{R}}$ requires $\Phi(\mathbf{l}, \mathbf{r})$ as a premise, and this premise is no longer available, we must also apply double negation to this part of the proof; the whole transformed proof now has as conclusion:

$$\forall l\,.\,\exists s\,.\,\neg\neg\Phi(l, s)$$

We could again implement the modified transformation as a simple rewrite rule, but it is more convenient to retain separate judgments for introducing double negations, and introduce subgoals into the tail-recursion transformation. The Elf code fragment that builds the base case of the induction subproof

```
(lforalli ([k] lexistsi _ (SO k) (lexistsi _ RO (andi Db (I_b k)))))
```

becomes

```
(lforalli [k] (lexistsi _ (SO k) (NN_I_b k)))
```

and we add the following subgoal to construct the proof of the double negation (which is schematic in **k**) and bind it to **NN_I_b**:

```
 <- ({k} doubleneg (lexistsi _ RO (andi Db (I_b k))) (NN_I_b k))
```

In the previous version of the transformation the step case of the induction is built by the following:

```
([l] [p] [x] (lforalli [k]
  (lexistse ([s] [q']
   (lexistse ([r] [q]
    lexistsi _ s
     (lexistsi _ (F x r)
       (andi (Ds x l r (andel q)) (I_s x k r s (ander q)))))
    q'))
  (lforalle (H x k) p))))
```

We change this to:

```
([l] [p] [x] lforalli [k]
  (lexistse ([s] [h]
   lexistsi _ s (NN_I_s l x k s h))
  (lforalle (H x k) p)))
```

Again we need a subgoal to construct the proof of the double negation, this time schematic in the parameters **x** ... **s** and the assumption **hyp**, and bind it to **NN_I_s**

```
 <- ({l} {x} {k} {s}
     doubleneg_1_prem
     ([hyp: |- (lexists ([r] (and (Phi l r) (Psi (H x k) r s))))]
      (lexistse ([r] [q: |- (and (Phi l r) (Psi (H x k) r s))]
       (lexistsi _ (F x r)
        (andi (Ds x l r (andel q)) (I_s x k r s (ander q)))))
       hyp))
     (NN_I_s l x k s))
```

The previous version of the transformation completes the proof via the following:

```
(lforalli ([l]
 lexistse ([s] [p']
  lexistse ([r] [p] (lexistsi _ s (I_r l r s p)))
   p')
  (lforalle KO
    (lforalle l
     %% New inductive sub-proof:
     ...
```

With double negation this becomes:

```
 (lforalli [l]
  (lexistse ([s] [p] lexistsi _ s (NN_I_r l s p))
   (lforalle KO
    (lforalle l
     %% New inductive sub-proof:
     ...
```

The following subgoal builds a proof of the required double negation, schematic in `l` and `s`, and binds it to **NN_I_r**

```
 <- ({l} {s}
     doubleneg_1_prem ([q] lexistse ([r] [q'] I_r l r s q') q) (NN_I_r l s)).
```

For the `select` example this transformation yields the following program:

Program 4.18

```
fun sel l =
 let fun sel' nil = (fn k => k)
       | sel' (x::l') = let val p = (sel' l') in
          fn k => (p (if x >= 2 then (x::k) else k)) end
 in (sel' l nil)
 end;
```

After the introduction of double negation the transformed proof proves $\forall l\,.\,\exists s\,.\,\neg\neg\Phi(l,s)$. Thus this is an example of specification transformation as well as proof transformation. In what sense then does the extracted program meet the original specification $\forall l\,.\exists s\,.\,\Phi(l,s)$? If the only admissible sense of the phrase "meets a specification" is "is extracted from a constructive proof of a specification", then it does not. But when Φ is not informative (as in the example) the term extracted from a proof of Φ has no computational use; that is the point of simplification during extraction. From the point of view of pure verification, $\neg\neg\Phi$ is an equally good specification. This loose argument is supported by the fact that other systems use closely related techniques to suppress computation. The PX system [Hay90] provides a modal operator, equivalent to double negation, for succinctly expressing classical truth. Sasaki [Sas86] uses double negation in the optimization of Nuprl programs, and also points out that other kinds of uninformative formulas can also be exploited to suppress computation. Paulin-Mohring [PM89] gives a type in the Calculus of Constructions for any A that hides its informative contents; this type is not quite the double negation of A but is a simpler type with similar properties. Schwichtenberg [Sch85] defines a classical existential quantifier equivalent to the double negation of the constructive one. He shows that his system is closed under the Markov rule by an analysis of normal derivations that is closely related to the double negation transformations we use.

Aside from these general considerations, although we have not carried out a proof, it is evident that the tail-recursion transformation with double negation yields an extracted program that is functionally equivalent to the transformation of Figure 4.5, if the predicate Φ is not informative. This follows from two considerations. First, extraction/simplification "ignores" uninformative subproofs, so the only computationally relevant expressions are those extracted from the witnessing term for $\exists s\,.\,\Phi(l,s)$; second, this witness is unaffected by the double negation introductions.

4.4 Discussion

We have shown how a well-known program transformation strategy, conversion of a recursive function definition to tail-recursive form, can be encoded and generalized as a proof transformation expressed as a rewrite rule.

This encoding provides two levels of correctness guarantees. At the program level, we know that the program meets its specification because it is extracted from a proof of the specification. This guarantee is inherent in the proofs-as-programs methodology, independent of implementation. At the level of proofs, the LF type system ensures that the transformed proof is a valid proof of a particular known end-formula. This is the same assurance that the object logic encoding gives for inference rules. As Harper et al. point out [HHP93], given the kind of encoding of natural deduction we have used, LF does not distinguish between inference rules and proofs of higher-order judgment type. Similarly, there is no distinction between inference rules and the proof transformation we have demonstrated here. It may be viewed as a derived rule of the logic, although one of an unusual kind: we normally think of an inference rule as taking proofs of some premises A_i and giving a proof of some new formula A. But the purpose of the proof transformation is to construct a new proof with a different structure, which may have the same end-formula as its input proof (as in the naive version of the tail-recursion transformation).

It may seem that our proof transformation is significantly different from a derived rule of inference because it analyses the structure of its input proof, whereas an inference rule depends only on the end-formula(s) of its input proof(s). This difference is an artifact of the way we have formulated the transformation. It is convenient to think of the transformation as taking a proof matching Proof Schema 4.5 as input because the writing of a proof of that form is an easy and natural first step in program development. But we could also conceive the transformation as the proof of a rather baroque derived rule of inference:

$$\dfrac{\boldsymbol{\Phi}([\,],\mathbf{r_0}) \qquad \begin{array}{c}\overline{\boldsymbol{\Phi}(l,r)}^{\,p}\\ \vdots \\ \boldsymbol{\Phi}(x :: l,\mathbf{f}(x,r))\end{array} \qquad \begin{array}{c}\overline{\boldsymbol{\Phi}(l,r) \wedge \boldsymbol{\Psi}(\mathbf{k_0},r,s)}^{\,p}\\ \vdots \\ \boldsymbol{\Phi}(l,s)\end{array} \qquad \begin{array}{c}\overline{\boldsymbol{\Psi}(\mathbf{h}(x,k),r,s)}^{\,p}\\ \vdots \\ \boldsymbol{\Psi}(k,\mathbf{f}(x,r),s)\end{array} \qquad \boldsymbol{\Psi}(k,\mathbf{r_0},\mathbf{s_0}(k))}{\forall l\,.\,\exists s\,.\,\neg\neg\boldsymbol{\Phi}(l,s)}\,p$$

From a formal point of view the only unusual feature of this rule is the imposition of constraints on individual terms represented by the occurrences of $\mathbf{r_0} \ldots \mathbf{h}$.

The ability to express the tail-recursion transformation as the proof of a derived rule of inference is significant because it results in a static, declarative, yet effective formulation that can be proved correct by the LF type system. This kind of transformation can be contrasted with proof transformations like cut-elimination (normalization in natural deduction systems) or the double-negation translations studied by Murthy [Mur90], which correspond to inference rules that are not derived, but admissible. The difference is significant for both the generality and the efficiency of the transformations. An admissible rule does not remain valid under all extensions of the logic in question, but a derived rule does. Thus a transformation expressible as a derived rule remains applicable as we work in different theories. Moreover, an admissible rule is usually proved by induction over the structure of proofs; as a result the complexity of its implementation is high. By restricting ourselves where possible to derived rules we obtain transformations that are valid for any theories we may wish to add to the core object logic, and that are feasible to implement. On the other hand we may also expect to be able to do more with an admissible rule that is not derived. Murthy's translations from classical to constructive logic are a notable example of a complex and correspondingly powerful proof of admissibility.

Chapter 5

Case Studies

In this chapter we turn from the discussion of techniques for encoding proof transformations to the application of these transformations to program development. We develop two case studies: breadth-first search in a tree, and depth-first search in a directed graph. These search procedures have applications in graph problems such as the detection of cycles, finding connected components, and topological sort.

In our development of breadth-first search we adapt the tail-recursion transformation of the previous chapter. As in the example of the `select` program, the transformation again induces a change of functionality in the extracted program. In our study of depth-first search we consider how proof transformation can be used to adapt to a small change in specification: we begin with a program to compute the transitive closure of a relation, and consider how to adapt its proof when the specification is changed to reflexive transitive closure. Here we develop a proof transformation closely related to the program transformation technique of *finite differencing* [PK80]. In both case studies, we examine the problem of removing function parameters that act as "loop bounds": they represent explicit termination proofs, but serve no purpose for computation.

5.1 Breadth-first search in a tree

Breadth-first search is a simple procedure often presented in terms of a queue and a while loop that terminates when the queue is empty [AHU83]. Termination of the loop is not quite straightforward to see, even in the absence of the complications introduced by the possibility of cycles in a general graph. This section considers the simplest possible case, that of a tree. We begin with a straightforward proof by induction on a measure of the size of the tree, which yields a function defined by primitive recursion over natural numbers. The recursion parameter is superfluous for the computation, so the next step in the development is to change the basis of the induction in order to remove this parameter. Finally, we transform the proof to tail-inductive form to obtain a function definition that can be compiled to the standard while-loop form of the algorithm.

5.1.1 Definitions

The usual theories of lists and natural numbers are assumed as a basis for the example. The syntax for lists given in Chapter 4 is extended by adding a definition of an append function for lists, denoted by $\circ$.

Definition 5.1 *(Labelled trees, forests)*

1. *A* labelled tree *is a structure* $\mathrm{node}(x,l)$ *where x (the label) is a natural number and l is a list of labelled trees.*
2. *A* forest *is a list of labelled trees.*

Definition 5.2 *$x \in_t t$ (x labels a node of t) if*

1. $t = \mathrm{node}(x,l)$, *or*
2. $t = \mathrm{node}(y,l)$, $x \neq y$, *and there is a tree u in l such that $x \in_t u$.*

We extend the notation to forests in the obvious way; $x \in_t l$ if there is a tree u in l such that $x \in_t u$.

A structural induction principle for labelled trees can be expressed by the following inference rule, parametric in x and l:

$$\dfrac{\begin{array}{c}\overline{\forall u\,.\,u \in l \supset [u/t]A}^{\,p}\\ \vdots \\ [\mathrm{node}(x,l)/t]A\end{array}}{\forall t\,.\,A}\;\mathrm{IND_T}^{p}$$

Definition 5.3 *The* size *of a tree is* $\mathrm{size}(\mathrm{node}(x,l)) = 1 + \sum_{t\in l}\mathrm{size}(t)$.

Again, we extend the notation to forests: $\mathrm{size}(l) = \sum_{t\in l}\mathrm{size}(t)$.

5.1.2 An initial algorithm

The traversal problem can be specified as one of collecting all the labels of a given tree.

Specification 5.4

$$\forall t\,.\,\exists r\,.\,\forall y\,.\,y \in r \Leftrightarrow y \in_t t$$

The method of proof determines the order of traversal. A proof by structural induction on trees leads to a depth-first traversal. For a breadth-first traversal we use induction on the size of a list of trees. Here is a sketch of an initial proof. (Recall that $\circ$ denotes the append function.)

Proof 5.5 We first generalize to a forest l and prove

Lemma 5.6 $\forall n\,.\,\forall l\,.\,\mathrm{size}(l) = n \supset (\exists r\,.\,\forall y\,.\,y \in r \Leftrightarrow y \in_t l)$

by induction on n. If $n = 0$, l must be the empty list, so choose $r = []$. Otherwise $n > 0$, and $l = \mathrm{node}(x, v) :: u$ for some x, v, and u. Since $\mathrm{size}(u \circ v) = n - 1$, by the induction hypothesis there is a list r' such that $\forall y \,.\, y \in r' \Leftrightarrow y \in_t u \circ v$. Then $x :: r'$ satisfies $\forall y \,.\, y \in x :: r' \Leftrightarrow y \in_t l$. □

Formalizing the proof and extracting a program requires adding a tree datatype with its size function to the object logic and programming language. We give an ML version of the necessary primitives:

```
datatype ''a tree = node of (''a * (''a tree) list)
fun size (node(x,l)) =
    1 + (fold (fn (a,b) => a+b) (map (fn a => size a) l) 0)
```

Then the following can be extracted from the proof:

Program 5.7

```
fun bfs t =
let fun bfs_aux 0 [] = []
      | bfs_aux 0 (t::l) = (any (neg ()))
      | bfs_aux n [] = (any (neg ()))
      | bfs_aux n ((node (x,v))::u) = (x::(bfs_aux (n-1) (u@v)))
in (bfs_aux (size t) [t]) end
```

Recall that program extraction produces `(any (neg ()))` from the inference $\perp \vdash C$ where C is arbitrary. This occurs in Proof 5.5 when the precondition $\mathrm{size}(l) = n$ is false.

5.1.3 Transforming the domain of induction

Because of the use of induction on natural numbers, Program 5.7 has an extraneous parameter `n`. This corresponds to a termination proof for the search. The parameter `n` can be eliminated after program extraction using Sasaki's *dead code unification* technique [Sas86]. Another approach is described for PX in [Hay90]; this can be adapted in an ad-hoc way to our setting. The advantage of this approach is that it operates at the level of proofs and simplifies further proof transformation.

PX provides *conditional inductive generations* to support the definition of inductively defined sets and the generation of inference rules for induction over these sets. The simple formalism presented here does not support this kind of definition, but at the meta-level one can introduce a new induction principle and prove that it is a derived rule by giving a transformation that expands an application of the new rule to a proof using only more primitive rules. The type-checking of an Elf encoding of the transformation corresponds to checking the derivation of the induction rule. Specifying how to extract code from the new inference rule also allows the transformation to be applied (in the other direction) to Proof 5.5 in order to obtain better extracted code. As Hayashi points out, this technique allows the separation of the termination proof of a program from other verification considerations; the derivation corresponds to the termination proof of a program development in the PX style.

The new induction principle needed, *tree list induction*, can be formulated as follows:

$$\dfrac{[\,[]/l]A \qquad \begin{array}{c}\overline{[u \circ v/l]A}^{\,p} \\ \vdots \\ [\mathrm{node}(x,v) :: u/l]A\end{array}}{\forall l \,.\, A}\ \mathrm{IND_{TS}}^{p}$$

The derivation of the rule follows from

1. $\forall l : \text{tree list} \,.\, \exists n \,.\, n = \mathrm{size}(l)$ is provable by list induction with a nested structural induction on trees.

2. $\forall n : \text{nat} \,.\, \forall l : \text{tree list} \,.\, n = \mathrm{size}(l) \supset A$ is provable by induction on natural numbers from the premises of $\mathrm{IND_{TS}}$ and the definition of size.

3. From (1) and (2) $\forall l : \text{tree list} \,.\, A$ follows trivially.

Using this derivation as a transformation on Proof 5.5 gives the following proof of Specification 5.4:

Proof 5.8 We generalize to a forest l and prove

Lemma 5.9 $\forall l \,.\, \exists r \,.\, \forall y \,.\, y \in r \Leftrightarrow y \in_t l$

by induction on size(l). When l is the empty list we choose $r = []$. Otherwise write l as $\mathrm{node}(x, v) :: u$. By the induction hypothesis there is a list r' such that $\forall y \,.\, y \in r' \Leftrightarrow y \in_t u \circ v$. Then $\forall y \,.\, y \in x :: r' \Leftrightarrow y \in_t l$. □

Code extraction for the new induction principle can be specified in the same style used for the core object logic in Section 2.3.2. We depart from that presentation by giving the extracted code in ML syntax, generating a function definition where f is a fresh variable name.

$$\dfrac{\mathrm{Inf}\ A \quad \langle \Gamma, \Delta \rangle \vdash \boxed{\begin{array}{c}\mathcal{P}_1 \\ [\,[]/l]A\end{array}} \Downarrow_s e_1 \qquad \begin{array}{c}\langle\, (\Gamma, x \Downarrow^i x', u \Downarrow^i u', v \Downarrow^i v'), \\ (\Delta, \boxed{\begin{array}{c}p \\ [u \circ v/l]A\end{array}} \Downarrow_s p')\, \rangle\end{array} \vdash \boxed{\begin{array}{c}p \\ [u \circ v/l]A \\ \mathcal{P}_2 \\ [\mathrm{node}(x,v) :: u/l]A\end{array}} \Downarrow_s e_2}{\langle \Gamma, \Delta \rangle \vdash \boxed{\dfrac{\begin{array}{c}\mathcal{P}_1 \\ [\,[]/l]A\end{array} \qquad \begin{array}{c}\overline{[u \circ v/l]A}^{\,p} \\ \mathcal{P}_2 \\ [\mathrm{node}(x,v) :: u/l]A\end{array}}{\forall l \,.\, A}\ \mathrm{IND_{TS}}^{p}} \Downarrow_s \ \begin{array}{l}\texttt{fun}\ f\ \texttt{[]}\ \texttt{=}\ e_1 \\ \ \ \texttt{|}\ f\texttt{(node(}x'\texttt{,}v'\texttt{)::}u'\texttt{)}\ \texttt{=} \\ \qquad [\texttt{(}f\ \texttt{(}u'\texttt{@}v'\texttt{))}/p']e_2\end{array}}\ \mathrm{xsIND_{TS}}$$

Then the following program can be extracted from Proof 5.8:

Program 5.10

```
fun bfs t =
let fun bfs_aux [] = []
      | bfs_aux ((node (x,v))::u) = (x::(bfs_aux (u@v)))
in (bfs_aux [t]) end
```

5.1.4 Adapting the tail-recursion transformation

Using the fold-unfold system Program 5.10 can be transformed to a tail-recursive definition by the same method shown in section 4.1.2 for the `select` example. Just as in that derivation, the resulting program accumulates its partial result by applying the append function. To obtain a tail-recursive program via proof transformation, we adapt the tail recursion transformation to tree list induction, and apply it to Proof 5.8.

In order to formulate proof transformations for a class of inductive proofs, it is useful to introduce some definitions. These are rather weak definitions, as it is not our purpose to formally treat induction in general in our simple implementation, but only to give an informal generalization which can easily be formalized for a given induction rule. An induction principle that meets Definition 5.11 is not thereby a valid one; it must be proved correct using the method of the previous section.

Definition 5.11 *A* simple induction principle *of the object logic is an inference rule of the form*

$$\frac{\mathcal{P}_1 \ldots \mathcal{P}_m}{\forall x_1 . \ldots \forall x_n . P \supset A}\,\mathrm{IND_S}^p$$

The precondition P may be empty, but must be Harrop if it is nonempty. Each premise $\mathcal{P}_i$ has the form:

1. $[t_{i1}, \ldots, t_{in}/x_1, \ldots, x_n]A$ *with each t_{ij} either a constant term or the parameter x_j, or*
2. $$\begin{array}{c} \overline{C_{i1}}^{\,p} \quad \ldots \quad \overline{C_{ik}}^{\,p} \qquad \overline{[d_{i1}, \ldots, d_{in}/x_1, \ldots, x_n]A}^{\,p} \\ \vdots \\ [t_{i1}, \ldots, t_{in}/x_1, \ldots, x_n]A \end{array}$$

   That is, $\mathcal{P}_i$ may depend on the premises C_{ij} and $[d_{i1}, \ldots, d_{in}/x_1, \ldots, x_n]A$ discharged by the application of the rule IND$_\mathrm{S}$. *Each C_{ij} is atomic.*

Given such an induction principle,

1. *A is the* induction predicate.
2. *P is the* precondition

3. $x_1 \dots x_n$ *are the* parameters of induction.

4. *A premise of the form (1) above is a* base case.

5. *A premise of the form (2) above is an* inductive case.

6. *For an inductive case* $\mathcal{P}_i$ *the terms* $d_{i1}, \dots, d_{in}$ *are the* descent terms.

7. *For an inductive case* $\mathcal{P}_i$ *any parameters occurring in* $C_{i1} \dots C_{ik}$*,* $d_{i1}, \dots, d_{in}$*, or* $t_{i1}, \dots, t_{in}$ *are the* proper parameters *of* $\mathcal{P}_i$.

For example, in tree list induction IND_{TS}, $x_1 \dots x_n$ corresponds to the single variable l of type tree list. The precondition is empty. The base case P_1 is $[\,[]/l]A$. The inductive case P_2 is $[\text{node}(x,v) :: u/l]A$, with a single discharged assumption $\{[u \circ v/l]A\}$, and proper parameters x, u, v.

Given a simple induction principle, a tail recursion transformation can be written for it. We describe it in terms of higher-order patterns using the notation of Section 4.1.3. The transformation is defined on an input proof that terminates in an application of the given induction rule with an induction predicate of the form $\exists r\,.\,\Phi(x_1, \dots, x_n, r)$, where $x_1, \dots, x_n$ are the parameters of induction. Each base case P_i has the following form, where $x_1, \dots, x_n$ may occur free in $\mathbf{t}_{i1}, \dots, \mathbf{t}_{in}$ (compare Proof schema 4.5):

Proof schema 5.12

$$\cfrac{\begin{matrix}\mathcal{D}_i \\ \vdots \\ \Phi(\mathbf{t}_{i1}, \dots, \mathbf{t}_{in}, \mathbf{r}_i)\end{matrix}}{\exists r\,.\,\Phi(\mathbf{t}_{i1}, \dots, \mathbf{t}_{in}, r)}\;\exists\text{I}$$

Each inductive case $\mathcal{P}_i$ has the following form, where the proper parameters $p_{i1}, \dots, p_{ij}$ of $\mathcal{P}_i$ may occur free in $\mathbf{C}_{i1} \dots \mathbf{C}_{ik}$, $\mathbf{d}_{i1}, \dots, \mathbf{d}_{in}$, and $\mathbf{t}_{i1}, \dots, \mathbf{t}_{in}$:

Proof schema 5.13

$$\cfrac{\exists r\,.\,\Phi(\mathbf{d}_{i1}, \dots, \mathbf{d}_{in}, r) \qquad \cfrac{\begin{matrix}\overline{\mathbf{C}_{i1}} \quad \dots \quad \overline{\mathbf{C}_{ik}} \qquad \overline{\Phi(\mathbf{d}_{i1}, \dots, \mathbf{d}_{in}, r)}^{\,q_i} \\ \vdots \\ \mathcal{D}_i \\ \vdots \\ \Phi(\mathbf{t}_{i1}, \dots, \mathbf{t}_{in}, \mathbf{f}_i(p_{i1}, \dots, p_{ij}, r))\end{matrix}}{\exists r\,.\,\Phi(\mathbf{t}_{i1}, \dots, \mathbf{t}_{in}, r)}\;\exists\text{I}}{\exists r\,.\,\Phi(\mathbf{t}_{i1}, \dots, \mathbf{t}_{in}, r)}\;\exists\text{E}^{q_i}$$

As in the special case of Chapter 4 the transformation requires an auxiliary specification Ψ expressing the way in which the result is to be accumulated. Thus the transformed proof contains a subproof, terminating in an application of the given induction rule, of

$$\forall x_1 \dots . \forall x_n\,.\,P \supset \forall k\,.\,\exists s\,.\,\exists r\,.\,\Phi(x_1, \dots, x_n, r) \wedge \Psi(k, r, s)$$

For ease of comparison the lemmas needed for the transformation are given here in the style of Section 4.1.3, although in an implementation the lifted forms described in Section 4.2 are easier to work with.

1. The transformation requires a proof $\mathcal{D}_R$ of

   Lemma schema 5.14 (Recovery lemma)

   $$\forall x_1 . \ldots \forall x_n . \forall r . \forall s . (\Phi(x_1, \ldots, x_n, r) \wedge \Psi(\mathbf{k}_0, r, s)) \supset \Phi(x_1, \ldots, x_n, s)$$

   for some term $\mathbf{k}_0$.

2. For each inductive case $\mathcal{P}_i$ of the subproof, the transformation requires a proof $\mathcal{I}_i$ of

   Lemma schema 5.15 (Inductive case lemma)

   $$\forall p_{i1} . \ldots \forall p_{ij} \forall k . \forall r . \forall s . \Psi(\mathbf{h}(p_{i1}, \ldots, p_{ij}, k), r, s) \supset \Psi(k, \mathbf{f}_i(p_{i1}, \ldots, p_{ij}, r), s) .$$

   where the pattern $\mathbf{f}_i(p_{i1}, \ldots, p_{ij}, r)$ *is given by the original proof and the pattern* $\mathbf{h}(p_{i1}, \ldots, p_{ij}, k)$ *is supplied by the user.*

3. For each base case we require a proof $\mathcal{I}_i$ of

   Lemma schema 5.16 (Base case lemma)

   $$\forall k . \Psi(k, \mathbf{r}_i, \mathbf{s}_i(k))$$

   $\mathbf{r}_i$ *is the witness term for the base case of the original proof;* $\mathbf{s}_i$ *is supplied by the user.*

The construction of the result of the transformation is analogous to the special case described in Chapter 4. For each base case $\mathcal{D}_i$ of the input proof the transformation constructs the following subproof from the original proof and the corresponding base case lemma. Here $\mathcal{I}'_i$ is derived from $\mathcal{I}_i$ by $\forall$E (it corresponds to the lifted form of Section 4.2). Double horizontal lines indicate the obvious sequence of introduction rules.

Proof schema 5.17

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_i\\ \vdots\\ \Phi(\mathbf{t}_{i1}, \ldots, \mathbf{t}_{in}, \mathbf{r}_i)\end{array} \qquad \begin{array}{c}\mathcal{I}'_i\\ \vdots\\ \Psi(k, \mathbf{r}_i, \mathbf{s}_i(k))\end{array}}{\Phi(\mathbf{t}_{i1}, \ldots, \mathbf{t}_{in}, \mathbf{r}_i) \wedge \Psi(k, \mathbf{r}_i, \mathbf{s}_i(k))}\wedge\mathrm{I}}{\overline{\forall k . \exists s . \exists r . \Phi(\mathbf{t}_{i1}, \ldots, \mathbf{t}_{in}, r) \wedge \Psi(k, r, s)}}$$

Similarly, for each inductive case $\mathcal{D}_i$ the following subproof can be constructed from the original proof and the corresponding inductive case lemma. Again $\mathcal{I}'_i$ corresponds to the lifted form of the proof $\mathcal{I}_i$ of Lemma Schema 5.15, and double lines indicate obvious introductions and eliminations.

Proof schema 5.18

$$\cfrac{\cfrac{\begin{array}{c}\overline{\mathbf{C}_{i1}} \quad \dots \quad \overline{\mathbf{C}_{ik}} \quad \overline{\Phi(\mathbf{d}_{i1},\dots,\mathbf{d}_{in},r)} \\ \vdots \\ \mathcal{D}_i \\ \vdots \\ \Phi(\mathbf{t}_{i1},\dots,\mathbf{t}_{in},\mathbf{f}_i(p_{i1},\dots,p_{ij},r))\end{array} \qquad \begin{array}{c}\overline{\overline{\Phi(\mathbf{d}_{i1},\dots,\mathbf{d}_{in},r) \ \wedge \ \Psi(\mathbf{h}(p_{i1},\dots,p_{ij},k),r,s)}} \\ \vdots \\ \mathcal{I}'_i \\ \vdots \\ \Psi(k,\mathbf{f}_i(p_{i1},\dots,p_{ij},r),s)\end{array}}{\Phi(\mathbf{t}_{i1},\dots,\mathbf{t}_{in},\mathbf{f}_i(p_{i1},\dots,p_{ij},r)) \ \wedge \ \Psi(k,\mathbf{f}_i(p_{i1},\dots,p_{ij},r),s)}\,\wedge\mathrm{I}}{\overline{\forall k\,.\,\exists s\,.\,\exists r\,.\,\Phi(\mathbf{t}_{i1},\dots,\mathbf{t}_{in},r) \ \wedge \ \Psi(k,r,s)}}$$

From these subproofs the transformation builds an induction in the obvious way. The original conclusion then follows from the recovery lemma $\mathcal{D}_R$ (again $\mathcal{D}'_R$ denotes the lifted form):

Proof schema 5.19

$$\begin{array}{c}\mathcal{D}'_{IND} \\ \vdots \\ \forall x_1\,.\,\dots\forall x_n\,.\,P \supset \forall k\,.\,\exists s\,.\,\exists r\,.\,\Phi(x_1,\dots,x_n,r) \ \wedge \ \Psi(k,r,s) \\ \hline\hline \Phi(x_1,\dots,x_n,r) \ \wedge \ \Psi(\mathrm{k}_0,r,s) \\ \vdots \\ \mathcal{D}'_R \\ \vdots \\ \Phi(x_1,\dots,x_n,s) \\ \hline\hline \forall x_1\,.\,\dots\forall x_n\,.\,P \supset \exists s\,.\,\Phi(x_1,\dots,x_n,s)\end{array}$$

Except for the difference in the induction principle, the transformation can be applied to the breadth-first search problem in the same way as it was to the list select example; the same auxiliary specification $\Psi(k,x,r) \equiv \forall y\,.\,y \in r \Leftrightarrow (y \in x \ \vee \ y \in k)$ can be used.

The new specification for the inductive subproof is:

Specification 5.20

$$\forall l\,.\,\forall k\,.\,\exists s\,.\,\exists r\,.\,(\forall y\,.\,y \in r \Leftrightarrow y \in_t l) \ \wedge \ (\forall y\,.\,y \in s \Leftrightarrow (y \in r \ \vee \ y \in k))$$

We show only the instantiation of the crucial lemma schema 5.15 that establishes the induction step. The term $\mathbf{h}_2(x,u,v,k)$ that governs the accumulation of the partial result is $x :: k$, and the term $\mathbf{f}_2(x,u,v,r)$ given by the original inductive subproof is $x :: r$. Thus the instantiation is (omitting the unused parameters u and v):

Lemma 5.21

$$\forall x\,.\,\forall k\,.\,\forall r\,.\,\forall s\,.\,[\forall y\,.\,y \in s \Leftrightarrow (y \in r \ \vee \ y \in x :: k)] \supset [\forall y\,.\,y \in s \Leftrightarrow (y \in x :: r \ \vee \ y \in k)]$$

Inserting the appropriate instantiations of the lemma schemas 5.14, 5.15, and 5.16 into the inductive subproof of Proof 5.8, results in a new proof of Specification 5.4. We sketch the proof here:

Proof 5.22 We first prove

$$\forall l . \forall k . \exists s . \exists r . (\forall y . y \in r \Leftrightarrow y \in_t l) \wedge (\forall y . y \in s \Leftrightarrow (y \in r \vee y \in k))$$

by induction on the tree list l. If $l = []$ choose $r = []$ and $s = k$. Otherwise $l = \text{node}(x, u) :: v$. By the induction hypothesis there are lists s' and r' such that $\forall y . y \in s' \Leftrightarrow (y \in r' \vee y \in x :: k)$ and $\forall y . y \in r' \Leftrightarrow y \in_t v \circ u$. Then let s be s' and let r be $x :: r'$.

Then for any tree t there are lists s and r such that $(\forall y . y \in r \Leftrightarrow y \in_t [t]) \wedge (\forall y . y \in s \Leftrightarrow (y \in r \vee y \in []))$. Then s contains exactly the labels of t. □

As in the example of Chapter 4, it is necessary to transform the proof further by inserting a double negation in order to eliminate the computation of r. The following program can then be extracted:

```
fun bfs t =
let fun bfs_aux [] k = k
      | bfs_aux ((node (x,v))::u) k = (bfs_aux (u@v) (x::k))
in (bfs_aux [t] []) end
```

5.1.5 Formalization

The formalization of tail-recursion introduction in Elf *for a given induction principle* is straightforward, following the procedure described in Chapter 4. Formalizing the transformation in general is more difficult and is beyond the scope of this thesis, requiring a general treatment of (restricted) induction. Ideally, the system would be augmented with a way of introducing an inductive definition (e.g., as in Nuprl, Coq or PX) with an associated rule of inference. An Elf logic program could then be written to analyze any proof by induction and apply the appropriate form of transformation.

Figure 5.1 shows the formalization of the syntax of trees and tree lists, with the functions append and size, equality for tree lists, and quantifiers.

The change of induction principle of Section 5.1.3 is formalized by giving a derivation in Elf of the induction principle IND$_{\text{TS}}$. The first step is to formulate IND$_{\text{TS}}$ as an inference rule:

```
indts : {A: tlist -> o}
        |- (A nl)
        -> ({X:i} {U:tlist} {V:tlist}
             |- (A (appnd U V)) -> |- (A (cns (nod X V) U)))
        -> |- (lforall A).
```

Then a pattern-driven proof transformation can be written to convert any proof using IND$_{\text{TS}}$ to one using induction on natural numbers, lists of trees, and trees, or vice-versa. Type-checking the clause defining the transformation verifies the derivation of the IND$_{\text{TS}}$ inference rule.

```
tre : type.                          %labelled trees
tlist : type.                        %lists of trees

nod : i -> tlist -> tre.             %tree constructor

nl : tlist.                          %empty tree list
cns : tre -> tlist -> tlist.         %cons for tree lists

appnd : tlist -> tlist -> tlist.     %append

size : tre -> i.                     %size of a tree
tlsize : tlist -> i.                 %size of a tree list

leq : tlist -> tlist -> o.           %equality for tree lists

tforall : (tre -> o) -> o.           %quantification over trees
texists : (tre -> o) -> o.

lforall : (tlist -> o) -> o.         %quantification over tree lists
lexists : (tlist -> o) -> o.
```

Figure 5.1: Elf encoding of trees and tree lists

As usual the transformation is encoded as a judgment and a single defining clause that expresses the rewrite. The judgment is:

```
expand_indts : {A: tlist -> o} |- (lforall A) -> |- (lforall A) -> type.
```

For readability we present the transformation in terms of the following constants defining lemmas needed for the derivation. This encoding hides the inductions on trees and lists of trees.

1. `p_nl : {L} |- (eq zero (tlsize L)) -> |- (leq nl L) -> type.`

   That is, for any list of trees l and proof that the size of l is zero, there is a proof that l is the empty list.

2.
   ```
   p_cns : {L} {N} |- (eq (succ N) (tlsize L))
           -> |- (exists [x] lexists [u] lexists [v]
                    (and (eq N (tlsize (appnd u v)))
                         (leq (cns (nod x v) u) L))) -> type.
   ```

   That is, for any l, n, and proof that the size of l is $n+1$, there is a proof that l can be decomposed as $\text{node}(x, u) :: v$ with $\text{size}(v \circ u) = n$.

3.
   ```
   p_ex : {L} |- (exists [n] eq n (tlsize L)) -> type.
   That is, for any list of trees $l$ there is a proof that $\siz(l)$ exists.
   ```

These lemmas are referenced as subgoals in the same way as is done for the double negation transformations of Section 4.3.

Figure 5.2 shows the Elf code for the rewrite rule.

5.2 Depth-first-search in a directed graph

This section presents an example of the use of proof transformation to adapt to a small change in specification. After some preliminary definitions and lemmas comes an initial proof and program to compute the transitive closure of a relation; if the relation is viewed as a directed graph, the program corresponds to a depth-first traversal of the graph. This proof is improved by changing the domain of induction using the method of Section 5.1.3. The resulting proof is adapted to compute the *reflexive* transitive closure by a process closely related to the finite differencing program transformation.

5.2.1 An initial algorithm

We begin with some basic notations and lemmas for graphs.

```
exp_indts :
  {A:tlist -> o} {Base_case:|- (A nl)}
  {Step_case:({X:i} {U:tlist} {V:tlist}
              |- (A (appnd U V)) -> |- (A (cns (nod X V) U)))}
  expand_indts A
   (indts A Base_case Step_case)
   (lforalli [l:tlist]
    existse
     ([n:i] [p: |- (eq n (tlsize l))]
      impliese
       (lforalle l (forall e n
        (ind ([n] lforall [l] implies (eq n (tlsize l)) (A l))
         (lforalli [l] impliesi [q: |- (eq zero (tlsize l))]
          (leq_subst_o ([l] A l) (P_nl l q) Base_case))
         ([n] [p': |- (lforall [l] implies (eq n (tlsize l)) (A l))]
          (lforalli [l] impliesi [q: |- (eq (succ n) (tlsize l))]
           (existse ([x] [p1] lexistse ([u] [p2] lexistse ([v]
            [p3: |- (and (eq n (tlsize (appnd u v)))
                         (leq (cns (nod x v) u) l))]
             (leq_subst_o ([l] A l)
              (ander p3)
              (Step_case x u v (impliese (lforalle (appnd u v) p') (andel p3))))
             p2) p1) (P_cns l n q)))))))
        p)
     (P_ex l))
  <- ({l} {q} p_nl l q (P_nl l q))
  <- ({l} {n} {q} p_cns l n q (P_cns l n q))
  <- ({l} p_ex l (P_ex l)).
```

Figure 5.2: Elf derivation of tree list induction

Definitions and lemmas

Definition 5.23 *A* graph *is a pair* $\langle V, E \rangle$ *where* V *is a finite enumerated set of nodes and* E *is a binary relation on* V.

Definition 5.24 *Given a relation* $E \in V \times V$, *a node* $x \in V$, *and a set* $U \subseteq V$, *we write*

1. xEy *when* $\langle x, y \rangle \in E$
2. $\hat{E}(x)$ *for* $\{y \mid xEy\}$
3. $\tilde{E}(U)$ *for* $\bigcup_{x \in U} \hat{E}(x)$

Definition 5.25 *Given a relation* $E \in V \times V$, *we define*

1. E^+, *the* transitive closure of E, *by*
   (a) xE^+y *when* xEy,
   (b) xE^+y *when for some* z xEz *and* zE^+y.
2. E^*, *the* reflexive transitive closure of E, *by*
   (a) xE^*x,
   (b) xE^*y *when for some* z xEz *and* zE^*y.

Lemma 5.26 *Given a relation* E *in* $V \times V$ *and* $U \subseteq V$, $\widetilde{E^*}(U) = U \cup \widetilde{E^+}(U)$.

Definition 5.27 *Given a graph* $\langle V, E \rangle$, *and nodes* $u, v \in V$, *a* path *in* $\langle V, E \rangle$ *from* u *to* v *is a sequence* $\langle x_0, x_1, \dots, x_n \rangle$ *with* $x_1, \dots, x_n \in V$ *and* $x_i E x_{i+1}$ *for* $1 \leq i < n$, *and* $x_0 = u$, $x_n = v$. *The* length *of the path is the number of edges* n.

The following definition models a common explanation (e.g. [AHU83]) of depth-first search, in which each node is marked "seen" as it is visited, and is thereafter excluded from the search in order to ensure termination. This can be modelled by restricting the codomain of the edge relation E to a given subset U of the node set V. Intuitively, this "cuts" edges into the set $V \setminus U$ but not the edges leaving the set. Operationally, U is the set of unmarked nodes.

Definition 5.28 *Given a binary relation* $E \in V \times V$, *and* $U \subseteq V$, $E_U = E \cap (V \times U)$. E_U^+ *denotes* $(E_U)^+$, *and similarly for* E_U^*.

Lemma 5.29 *Given a graph* $\langle V, E \rangle$, *sets* $R, U \subseteq V$, $U' \subseteq U$, *and nodes* $x, y \in V$, $u \in U$:

1. $\widetilde{E_{U'}^+}(R) \subseteq \widetilde{E_U^+}(R)$ *and* $\widetilde{E_{U'}^*}(R) \subseteq \widetilde{E_U^*}(R)$.
2. $x(E_{U \setminus \{u\}})^*y$ *if and only if there is a path* p *from* x *to* y *such that if* u *occurs in* p *then* $u = x$.

3. $x(E_{U\setminus\{u\}})^+y$ *if and only if there is a path p of length $n > 0$ from x to y such that if u occurs in p then $u = x$.*

The following lemma is crucial to depth-first search since the algorithm depends on the elimination of cycles for termination.

Lemma 5.30 (Cycle Lemma) *Given a graph $\langle V, E \rangle$, a set $U \subseteq V$, nodes $x, y \in V$, and $u \in U$, if $x(E_U)^*y$ then $x(E_{U\setminus\{u\}})^*y$ or there is a node z such that $x(E_{U\setminus\{u\}})^*z$ and zEu and $u(E_{U\setminus\{u\}})^*y$.*

Proof 5.31 Assume $u \in U$ and xE_U^*y. Then there is a path $p = \langle x_0, \ldots, x_n \rangle$ with $n \geq 0$, $x_0 = x$, $x_n = y$, $x_i \in U$ for $i > 0$, and x_iEx_{i+1} for $0 \leq i < n$. If u does not occur in p then $x(E_{U\setminus\{u\}})^*y$ trivially. Otherwise let x_j be the first occurrence of u in p and let x_k be the last occurrence of u in p.

If $j = 0$ (i.e. $u = x = x_0$) then $\langle u, x_k + 1, \ldots, x_n \rangle$ is a path from x to y; thus $x(E_{U\setminus\{u\}})^*y$.

Otherwise if $j = n$ (i.e. $u = y = x_n$) then $\langle x_0, \ldots, x_{j-1}, u \rangle$ is a path from x to y. Then $x(E_{U\setminus\{u\}})^*x_{j-1}$, $x_{j-1}Eu$, and $u(E_{U\setminus\{u\}})^*y$.

Otherwise, $\langle x_0, \ldots, x_{j-1}, u, x_{k+1}, \ldots, x_n \rangle$ is a path from x to y. Then $x(E_{U\setminus\{u\}})^*x_{j-1}$, $x_{j-1}Eu$, and $u(E_{U\setminus\{u\}})^*y$.

Corollary: For $x, y \in V$ and $u \in U$, if $x(E_U)^+y$ then $x(E_{U\setminus\{u\}})^+y$ or there is a node z such that $x(E_{U\setminus\{u\}})^+z$ and zEu and $u(E_{U\setminus\{u\}})^*y$. □

A proof and program for transitive closure

Depth-first search in a directed graph arises from induction on a pair of sets of nodes $\langle U, R \rangle$. U is the set of nodes to be searched and R is a set of root nodes from which the search begins.

Here is a specification for computing the set of nodes reachable along a nonempty path from a given set of nodes R in a directed graph $\langle V, E \rangle$:

Specification 5.32 $\forall \langle V, E \rangle : \text{graph}, R \,.\, R \subseteq V \supset \exists W \,.\, W = \widetilde{E^+}(R)$

The proof proceeds by generalization to subsets U of V and to the relation E_U:

Lemma 5.33 $\forall \langle V, E \rangle : \text{graph}, U, R \,.\, U, R \subseteq V \supset \exists W \,.\, W = \widetilde{E_U^+}(R)$

Specification 5.32 follows easily with $U = V$.

Proof 5.34 (of Lemma 5.33) The proof is by induction on the pair of sets $\langle U, R \rangle$. At each step of the induction we decrease U or hold U constant and decrease R.

Case $U = \{\}$ or $R = \{\}$; then let $W = \{\}$.

Case $U \neq \{\}$ and $R = r \uplus R'$. There are two cases:

1. $\widetilde{E_U}(r) = \{\}$. Then apply the induction hypothesis to U, R' to obtain $W' = \widetilde{E_U^+}(R')$.

   Claim: $W' = \widetilde{E_U^+}(R)$. Write W for $\widetilde{E_U^+}(R)$. By Lemma 5.29 $W' \subseteq W$. Suppose $x \in W$; then for some $r' \in R$, $r'(E_U)^+x$. If $r' \in R'$ then trivially $x \in W'$. But r' cannot be r, for then there would be a node u in U with rEu, contradicting $\widetilde{E_U}(r) = \{\}$. So $W \subseteq W'$.

2. $\widehat{E_U}(r) = s \uplus S$. Write U' for $U \setminus \{s\}$ and apply the induction hypothesis to $U', R \cup \{s\}$ to obtain $W' = \widetilde{E_{U'}^+}(R \cup \{s\})$.

   Claim: $\{s\} \cup W' = \widetilde{E_U^+}(R)$. Again write W for $\widetilde{E_U^+}(R)$. Suppose $x \in W$, that is, $r' E_U^+ x$ for some $r' \in R$. By the corollary to Lemma 5.30 there are two cases:

   (a) $r'(E_{U'})^+ x$. Then $x \in W'$ since $r' \in R \cup \{s\}$.

   (b) There is a node z such that $r'(E_{U'})^+ z$ and zEs and $s(E_{U'})^* x$. If $x = s$ then $x \in \{s\} \cup W'$ trivially. Otherwise $s(E_{U'})^+ x$ and since $s \in R \cup \{s\}$, $x \in W'$.

   So $W \subseteq s \cup W'$.

   For the other direction of set inclusion, suppose $x \in W'$, that is $r'(E_{U'})^+ x$ for some $r' \in R \cup \{s\}$. If $r' \in R$ the inclusion follows easily. If $r' = s$ then since rEs and $s \in U$ it follows that $r(E_U)^+ x$, so $x \in W$. So $W' \subseteq W$. Clearly $s \in W$ since rEs and $s \in U$.

□

The proof is formalized as a proof by nested induction on the size of the set U and the structure of the set R.

In order to formalize the proof and extract a program it is necessary to extend the object logic and programming language with a sort of finite enumerated sets with a structural induction principle, and the operations of disjoint union with a singleton, the deletion of a single element, intersection, and cardinality. On this sort is built another for graphs. A graph is represented as a pair of a set of nodes and a finite mapping that takes a node to the set of adjacent nodes. The object logic under consideration forces a fixed choice of data type for representing nodes but this does not affect the transformation process. Note that there are no set comprehensions in the formal language; the notations used in the foregoing definitions should be considered as abbreviations.

Programs using these data types are presented in an ML-like syntax extended as follows:

e	::=	... | **empty** | e_1++e_2 |	*Finite sets: empty set, singleton union*
		e_1||e_2 | e_1&&e_2 | e_1--e_2 | crd$e$ |	*Union, intersection, deletion, cardinality*
		graph(e_1,e_2) | **mapp**(e_1,e_2) |	*Graphs*

The constructor ++ is analogous to **cons** for lists; an expression of the form e_1 ++ e_2 constructs the union of a singleton containing the element e_1 with a set e_2. The expression e_1 -- e_2 denotes the set obtained by deleting the element e_2 from the set e_1. The union and intersection of two sets are represented by || and && respectively. The size of a set e is obtained by crd e.

An expression **graph(V,E)** constructs a representation of a graph $\langle V, E \rangle$; an expression **mapp(x,G)** evaluates to the set of nodes adjacent to the node **x**; that is, it represents the computation of $\widehat{E}(x)$ for a graph $G = \langle V, E \rangle$. As usual we freely introduce bindings to improve readability of programs.

The argument and call structure of the extracted program is a consequence of the structure of the formalization of Proof 5.34. Here is a sketch of that structure:

First Lemma 5.33 is formalized as

Lemma 5.35

$\forall\langle V, E\rangle . \forall n . \forall U . (U \subseteq V \wedge n = |U|) \supset \forall R . R \subseteq V \supset \exists W . \forall y . y \in W \iff \exists x . x \in R \wedge x E_U^+ y$

This presentation still takes some liberty with notation: the use of destructuring is not supported in the object logical language. However for readability we write $\langle V, E\rangle$, V, and E rather than G, **fst**(G), and **snd**(G).

Proof outline 5.36 The proof is by induction on n. If $n = 0$ then $U = \{\}$; let $W = \{\}$.

If $n = n' + 1$ then U is nonempty, and we show

$$\forall R . U, R \subseteq V \supset \exists W . \forall y . y \in W \iff \exists x . x \in R \wedge x E_U^+ y$$

by induction on the structure of R. If $R = \{\}$ then let $W = \{\}$.

Otherwise $R = r \uplus R'$. There are two cases:

1. $\widehat{E_U}(r) = \{\}$. Then apply the inner induction hypothesis to R' to obtain W' and let $W = W'$.
2. $\widehat{E_U}(r) = s \uplus S$. Write U' for $U \setminus \{s\}$ and apply the outer induction hypothesis to U'. Instantiate R as $R \cup \{s\}$ to obtain W', and let $W = \{s\} \cup W'$.

□

The extracted program:

Program 5.37

```
fun tclos (G as (graph(V,_))) R =
let fun tclos_1 0 empty R = empty
      | tclos_1 0 (u ++ U) R = raise neg
      | tclos_1 n U R =
  let fun tclos_2 empty = empty
        | tclos_2 (R as (r ++ R')) =
           case (mapp(r, G) && U) of
             empty => (tclos_2 R')
           | (s ++ S) =>
              (s ++ (tclos_1 (n-1) (U -- s) (s ++ R)))
  in (tclos_2 R) end
in (tclos_1 (crd V) V R) end
```

Although the asymptotic complexity of this program is already good ($|V|^2$, assuming appropriate implementations of the set operations), just as in the breadth-first search program (Program 5.7), the induction on natural numbers results in an unwanted parameter for the function `tclos_1`. The nested set induction causes the extraction of nested mutually recursive function definitions which can be combined into one definition. The next section shows how to apply the technique of Section 5.1.3 to solve both these problems.

$$\dfrac{[\{\}/U]A \qquad [\{\}/R]A \qquad \begin{array}{c}\overline{\tilde{E}(r) \cap U = \{\}, A}^{\,p}\\ \vdots \\ [r \uplus R/R]A\end{array} \qquad \begin{array}{c}\overline{\tilde{E}(r) \cap U = s \uplus S, [U \setminus \{s\}/U][r \uplus R \cup \{s\}/R]A}^{\,p}\\ \vdots \\ [r \uplus R/R]A\end{array}}{\forall \langle V, E \rangle . \forall U . \forall R . (U \subseteq V \;\wedge\; R \subseteq V) \supset A}\;\mathrm{IND_{DF}}^{p}$$

Figure 5.3: Induction principle for depth-first search

5.2.2 Transforming the domain of induction

The technique applied here is the same as that applied to the breadth-first search problem: introduce a new induction rule and give its derivation from more primitive inference rules; encoded in Elf, the derivation can be executed as a program transformation. However, here the new rule $\mathrm{IND_{DF}}$ is more complex. It is shown in Figure 5.3. We sketch its derivation:

Theorem 5.38 *Given a predicate A in which variables V, E, U, R may occur free, the rule* $\mathrm{IND_{DF}}$ *is derivable in the object logic with induction on natural numbers and structural induction on finite sets.*

Proof 5.39 Under the given assumptions, a proof of

$$\forall \langle V, E \rangle . \forall n . \forall U . (U \subseteq V \;\wedge\; n = |U|) \supset \forall R . R \subseteq V \supset A$$

can be constructed by nested inductions on n and the structure of R as in Proof outline 5.36.

Since the size of a set is primitive in the object logic, $\forall U . \exists n . n = |U|$ is provable. The required proof can be constructed by the elimination rule for the existential quantifier, with trivial introductions and eliminations for implication and the universal quantifier. □

We sketch the extraction process for the derived rule here without giving all the details. For readability the extracted expression is shown in the extended ML-like syntax given above, using recursive function definition and pattern matching.

Given extraction assumptions $\langle \Gamma, \Delta \rangle$, and an object proof of the following form:

$$\dfrac{\begin{array}{c}\mathcal{P}_1\\ [\{\}/U]A\end{array} \qquad \begin{array}{c}\mathcal{P}_2\\ [\{\}/R]A\end{array} \qquad \begin{array}{c}\overline{\tilde{E}(r) \cap U = \{\}, A}^{\,p}\\ \mathcal{P}_3 \\ [r \uplus R/R]A\end{array} \qquad \begin{array}{c}\overline{\tilde{E}(r) \cap U = s \uplus S, [U \setminus \{s\}/U][r \uplus R \cup \{s\}/R]A}^{\,p}\\ \mathcal{P}_4 \\ [r \uplus R/R]A\end{array}}{\forall \langle V, E \rangle . \forall U . \forall R . (U \subseteq V \;\wedge\; R \subseteq V) \supset A}\;\mathrm{IND_{DF}}^{p}$$

extract an object program of the following form:

```
fun f G empty R = e1
  | f G U empty = e2
  | f G U (R as (r++R')) =
    case (mapp(r,G) && U) of
     empty => [(f G U R')/p]e3
     (s++S) => [(f G (U -- s) (s++R))/p]e4
```

The subexpressions $e_1 \ldots e_4$ are given by the following premises for the extraction. The notations $\Downarrow^g$ and $\Downarrow^S$, and indicate the extraction judgments for graphs and finite sets, respectively. Different fonts distinguish variables of the logic (x) from variables of the programming language ($\mathtt{x}$).

- Inf A (A is informative)

- $\langle\, (\Gamma, G \Downarrow^g \mathtt{G}, R \Downarrow^S \mathtt{R}), \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P}_1 \\ [\{\}/U]A \end{array}} \Downarrow_s e_1$

- $\langle\, (\Gamma, G \Downarrow^g \mathtt{G}, U \Downarrow^S \mathtt{U}), \Delta \,\rangle \vdash \boxed{\begin{array}{c} \mathcal{P}_2 \\ [\{\}/R]A \end{array}} \Downarrow_s e_2$

- $\langle\, (\Gamma, G \Downarrow^g \mathtt{G}, U \Downarrow^S \mathtt{U}, R \Downarrow^S \mathtt{R}, r \Downarrow^i \mathtt{r}, S \Downarrow^S \mathtt{S}, s \Downarrow^i \mathtt{s}), (\Delta, \boxed{\overset{p}{E(r) \cap U = \{\} \ \wedge \ A}} \Downarrow_s \mathtt{p}) \,\rangle$
  $\vdash \boxed{\begin{array}{c} p \\ E(r) \cap U = \{\} \ \wedge \ A \\ \mathcal{P}_3 \\ [r \uplus R/R]A \end{array}} \Downarrow_s e_3$

- $\langle\, (\Gamma, G \Downarrow^g \mathtt{G}, U \Downarrow^S \mathtt{U}, R \Downarrow^S \mathtt{R}, r \Downarrow^i \mathtt{r}, S \Downarrow^S \mathtt{S}, s \Downarrow^i \mathtt{s}),$
  $(\Delta, \boxed{\overset{p}{E(r) \cap U = s \uplus S \ \wedge \ [U \setminus \{s\}/U][r \uplus R \cup \{s\}/R]A}} \Downarrow_s \mathtt{p}) \,\rangle$
  $\vdash \boxed{\begin{array}{c} p \\ E(r) \cap U = s \uplus S \ \wedge \ [U \setminus \{s\}/U][r \uplus R \cup \{s\}/R]A \\ \mathcal{P}_4 \\ [r \uplus R/R]A \end{array}} \Downarrow_s e_4$

With the derivation of IND$_{\text{DF}}$ we can transform Proof 5.36 to a similar proof based on the new induction rule IND$_{\text{DF}}$. The extraction procedure sketched above yields a new program that improves on Program 5.37 in two ways. The extraneous termination parameter has been removed. In addition, because the derived induction rule corresponds to two nested inductions, the new program contains only one local recursive definition instead of two.

Program 5.40

```
fun tclos' (G as (graph(V,E))) R =
let fun tclos_1' G empty R = empty
      | tclos_1' G U empty = empty
      | tclos_1' G U (R as (r++R')) =
           case (mapp(r,G) && U) of
             empty => (tclos_1' G U R')
           | (s++S) =>
              (s ++ (tclos_1' G (U -- s) (s++R)))
in (tclos_1' G V R) end
```

5.2.3 A transformation to reflexive transitive closure

Lemma 5.26, which says that $\widetilde{E^*}(R) = \widetilde{E^+}(R) \cup U$ for any $R \subseteq V$, is the basis for an easy proof for reflexive transitive closure. This proof contains Proof 5.34 as a subproof.

Specification 5.41 $\forall\langle V, E\rangle : \text{graph}, R . R \subseteq V \supset \exists X . X = \widetilde{E^*}(R)$

The proof proceeds by generalization in the same way as for the transitive closure Specification 5.32:

Lemma 5.42 $\forall\langle V, E\rangle : \text{graph}, U, R . U, R \subseteq V \supset \exists X . X = R \cup \widetilde{E^+_U}(R)$

Specification 5.41 follows easily with $U = V$ by Lemma 5.26.

Proof 5.43 (of Lemma 5.42) The proof is trivial: by Lemma 5.33 there is a set $W = \widetilde{E^+_U}(R)$; form the set $R \cup W$. □

The extracted program relies on the local function definition `tclos_1'` of Program 5.40, extracted from Proof 5.34:

Program 5.44

```
fun rtclos (G as (graph(V,E))) R =
let fun tclos_1 G empty R = empty
      | tclos_1 G U empty = empty
      | tclos_1 (G as graph(V,E)) U (R as (r++R')) =
           case ((mapp(r,G)) && U) of
             empty => (tclos_1 G U R')
           | (s++S) =>
              (s ++ (tclos_1 G (U -- s) (s++R)))
in R || (tclos_1 G V R) end
```

A finite differencing transformation

The transformation problem in this example is a special case of finite differencing [PK80]. Proof 5.43 gives the reflexive transitive closure of a set R as the union of R and its transitive closure; finite differencing can be used to compute the union incrementally. As a program transformation, finite differencing applies when there is a ***difference equation*** expressing incremental computation, such as $(A \cup \{a\}) \cup B = (A \cup B) \cup \{a\}$. This section shows how to use such equations at the level of proofs in an analogous way. The transformation is very similar to the tail-recursion transformation; in fact, the analysis of the inductive subproof is identical to the analysis for tail-recursion introduction. This is not surprising since both are instances of promotion, as pointed out in [Bir84].

The problem in this particular form of finite differencing is to distribute the computation of set union over a recursive function definition. A proof transformation for this purpose can be formulated for any simple induction principle IND_S (Definition 5.11). It operates on a proof containing an inductive subproof and, like the tail-recursion transformation, requires proofs of lemmas for each case of the induction. Since it is based on equations, it is somewhat simpler to specify. As usual we describe it in terms of higher-order patterns.

In what follows we abbreviate the parameters and terms of the given induction principle by tuple variables. Thus $\bar{x}_n$ denotes the parameters of induction $x_1, \ldots, x_n$, and $\bar{\mathbf{d}}_{in}$ denotes the terms $\mathbf{d}_{i1}, \ldots, \mathbf{d}_{in}$ occurring in an induction hypothesis, etc.

The input proof has the form

Proof schema 5.45

$$\begin{array}{c} \mathcal{D}_{IND} \\ \vdots \\ \forall \bar{x}_n . \mathbf{P} \supset \exists z . \Phi(\bar{x}_n, z) \\ \hline\hline \Phi(\bar{x}_n, z) \\ \vdots \\ \mathcal{D} \\ \vdots \\ \Psi(\bar{x}_n, \mathbf{g}(\bar{x}_n, z)) \\ \hline\hline \forall \bar{x}_n . \mathbf{P} \supset \exists y . \Psi(\bar{x}_n, y) \end{array}$$

where $\mathcal{D}_{IND}$ terminates in an application of the simple induction rule IND_S. As usual, double horizontal lines abbreviate sequences of obvious introduction or elimination inferences. We will use the subproof $\mathcal{D}$ to form an auxiliary specification for a new inductive subproof in the style of the tail recursion transformation. The transformed induction will prove

$$\forall \bar{x}_n . \mathbf{P} \supset \exists y . \neg\neg\exists z . \Phi(\bar{x}_n, z) \wedge y = \mathbf{g}(\bar{x}_n, z)$$

However, the presentation follows the same format as that for the tail-recursion transformation: first we describe a transformation that produces an inductive subproof without double negation. The double negation transformations of Section 4.3 can then be applied to selected subproofs of the result to eliminate unwanted computation.

We repeat the patterns describing the inductive subproof of the input proof from Section 5.1.4, changing some parameter names for convenience.

Each base case of $\mathcal{D}_{IND}$ has the following form, where $x_1, \ldots, x_n$ may occur free in $\mathbf{t}_{i1}, \ldots, \mathbf{t}_{in}$:

Proof schema 5.46

$$\dfrac{\begin{array}{c}\mathcal{B}_i \\ \vdots \\ \mathbf{\Phi}(\bar{\mathbf{t}}_{in}, \mathbf{f}_i(\bar{x}_n))\end{array}}{\exists z\,.\,\mathbf{\Phi}(\bar{\mathbf{t}}_{in}, z)}\,\exists\mathrm{I}$$

Each inductive case $\mathcal{P}_i$ has the following form, where the proper parameters $p_{i1}, \ldots, p_{ij}$ of $\mathcal{P}_i$ may occur free in $\mathbf{C}_{i1} \ldots \mathbf{C}_{ik}$, $\mathbf{d}_{i1}, \ldots, \mathbf{d}_{in}$, and $\mathbf{t}_{i1}, \ldots, \mathbf{t}_{in}$:

Proof schema 5.47

$$\dfrac{\exists z\,.\,\mathbf{\Phi}(\bar{\mathbf{d}}_{in}, z) \qquad \dfrac{\begin{array}{c}\overline{\mathbf{C}_{i1}} \quad \ldots \quad \overline{\mathbf{C}_{ik}} \qquad \overline{\mathbf{\Phi}(\bar{\mathbf{d}}_{in}, z)}^{\,q_i} \\ \vdots \\ \mathcal{I}_i \\ \vdots \\ \mathbf{\Phi}(\bar{\mathbf{t}}_{in}, \mathbf{f}_i(\bar{p}_{ij}, z))\end{array}}{\exists z\,.\,\mathbf{\Phi}(\bar{\mathbf{t}}_{in}, z)}\,\exists\mathrm{I}}{\exists z\,.\,\mathbf{\Phi}(\bar{\mathbf{t}}_{in}, z)}\,\exists\mathrm{E}^{q_i}$$

For each case $\mathcal{P}_i$ of the induction the transformation requires a proof $\mathcal{L}_i$ of an equation. For a base case $\mathcal{L}_i$ must prove, for some $\mathbf{h}_i$ and for all $\bar{x}_n$:

$$\mathbf{h}_i(\bar{x}_n) = \mathbf{g}(\bar{\mathbf{t}}_{in}, \mathbf{f}_i(\bar{x}_n))$$

For an inductive case $\mathcal{L}_i$ must prove, for some $\mathbf{h}_i$ and for all $\bar{p}_{ij}$ and z:

$$\mathbf{h}_i(\bar{p}_{ij}, \mathbf{g}(\bar{\mathbf{d}}_{in}, z)) = \mathbf{g}(\bar{\mathbf{t}}_{in}, \mathbf{f}_i(\bar{p}_{ij}, z))$$

This equation is used with the induction hypothesis when constructing an inductive case of the output proof to provide a witness t for $\exists y\,.\,y = \mathbf{g}(\bar{\mathbf{t}}_{in}, \mathbf{f}_i(\bar{p}_{ij}, z))$. The witness term is $\mathbf{h}_i(\bar{p}_{ij}, y)$ where y is, in turn, the witness provided by the induction hypothesis. Thus $\mathbf{h}_i$ should be chosen so that it is cheap to compute, compared to the cost of $\mathbf{g}(\bar{x}_n, z)$. For the transitive closure problem, assuming a cost of $(|A|+|B|)^2$ for $A \cup B$, the cost of $\mathbf{h}_i$ should be no more than a constant since the induction principle results in a $|V|^2$ algorithm. As expected (since depth-first search is inherently $|V|^2$ in time complexity) for this example the transformation does not produce an asymptotic improvement in the execution time of the algorithm.

The transformation yields a proof of the original specification, containing an inductive subproof of

$$\forall \bar{x}_n\,.\,\mathbf{P} \supset \exists y\,.\,\exists z\,.\,\mathbf{\Phi}(\bar{x}_n, z) \ \wedge\ y = \mathbf{g}(\bar{x}_n, z)$$

The cases of the induction are constructed from the cases of the original induction subproof and the lemmas $\mathcal{L}_i$.

Each base case of the induction is constructed as follows:

Proof schema 5.48

$$\dfrac{\dfrac{\begin{matrix}\mathcal{B}_i\\ \vdots\\ \mathbf{\Phi}(\bar{\mathbf{t}}_{in},\mathbf{f}_i(\bar{x}_n))\end{matrix}\qquad\qquad \begin{matrix}\mathcal{C}_i\\ \vdots\\ \mathbf{h}_i(\bar{x}_n)=\mathbf{g}(\bar{\mathbf{t}}_{in},\mathbf{f}_i(\bar{x}_n))\end{matrix}}{\mathbf{\Phi}(\bar{\mathbf{t}}_{in},\mathbf{f}_i(\bar{x}_n))\wedge \mathbf{h}_i(\bar{x}_n)=\mathbf{g}(\bar{\mathbf{t}}_{in},\mathbf{f}_i(\bar{x}_n))}{\scriptstyle \wedge\mathrm{I}}}{\exists y\,.\,\exists z\,.\,\mathbf{\Phi}(\bar{\mathbf{t}}_{in},z)\wedge y=\mathbf{g}(\bar{\mathbf{t}}_{in},z)}{\scriptstyle \exists\mathrm{I}}$$

Each inductive case has the form:

Proof schema 5.49

$$\dfrac{\exists y\,.\,\exists z\,.\,\mathbf{\Phi}(\bar{\mathbf{d}}_{in},z)\wedge y=\mathbf{g}(\bar{\mathbf{d}}_{in},z)\qquad \dfrac{\dfrac{\begin{matrix}\mathbf{C}_{i1}\ldots\mathbf{C}_{ik}\quad \dfrac{}{\mathbf{\Phi}(\bar{\mathbf{d}}_{in},z)}{\scriptstyle q_i,\wedge\mathrm{E}_L}\\ \vdots\\ \mathcal{I}_i\\ \vdots\\ \mathbf{\Phi}(\bar{\mathbf{t}}_{in},\mathbf{f}_i(\bar{p}_{ij},z))\end{matrix}\qquad \dfrac{\dfrac{}{y=\mathbf{g}(\bar{\mathbf{d}}_{in},z)}{\scriptstyle q_i,\wedge\mathrm{E}_R}\qquad \begin{matrix}\mathcal{C}_i\\ \vdots\\ \mathbf{h}_i(\bar{p}_{ij},\mathbf{g}(\bar{\mathbf{d}}_{in},z))\\ =\mathbf{g}(\bar{\mathbf{t}}_{in},\mathbf{f}_i(\bar{p}_{ij},z))\end{matrix}}{\mathbf{h}_i(\bar{p}_{ij},y)=\mathbf{g}(\bar{\mathbf{t}}_{in},\mathbf{f}_i(\bar{p}_{ij},z))}{\scriptstyle =\mathrm{S}}}{\mathbf{\Phi}(\bar{\mathbf{t}}_{in},\mathbf{f}_i(\bar{p}_{ij},z))\wedge \mathbf{h}_i(\bar{p}_{ij},y)=\mathbf{g}(\bar{\mathbf{t}}_{in},\mathbf{f}_i(\bar{p}_{ij},z))}{\scriptstyle \wedge\mathrm{I}}}{\exists y\,.\,\exists z\,.\,\mathbf{\Phi}(\bar{\mathbf{t}}_{in},z)\wedge y=\mathbf{g}(\bar{\mathbf{t}}_{in},z)}{\scriptstyle \exists\mathrm{I}}}{\exists y\,.\,\exists z\,.\,\mathbf{\Phi}(\bar{\mathbf{t}}_{in},z)\wedge y=\mathbf{g}(\bar{\mathbf{t}}_{in},z)}{\scriptstyle \exists\mathrm{E}^{q_i}}$$

The new proof has the form:

Proof schema 5.50

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{\Phi}(\bar{x}_n,z)\wedge y=\mathbf{g}(\bar{x}_n,z)}{\scriptstyle a}}{y=\mathbf{g}(\bar{x}_n,z)}{\scriptstyle \wedge\mathrm{E}_R}\qquad \begin{matrix}\dfrac{\dfrac{}{\mathbf{\Phi}(\bar{x}_n,z)\wedge y=\mathbf{g}(\bar{x}_n,z)}{\scriptstyle a}}{\mathbf{\Phi}(\bar{x}_n,z)}{\scriptstyle \wedge\mathrm{E}_L}\\ \vdots\\ \mathcal{D}\\ \vdots\\ \mathbf{\Psi}(\bar{x}_n,\mathbf{g}(\bar{x}_n,z))\end{matrix}}{\mathbf{\Psi}(\bar{x}_n,y)}{\scriptstyle =\mathrm{S}}}{\mathbf{\Phi}(\bar{x}_n,z)\wedge y=\mathbf{g}(\bar{x}_n,z)\supset\mathbf{\Psi}(\bar{x}_n,y)}{\scriptstyle \supset\mathrm{I}^a}\qquad \dfrac{\begin{matrix}\mathcal{D}'_{IND}\\ \vdots\\ \forall\bar{x}_n\,.\,\mathbf{P}\supset\exists y\,.\,\exists z\,.\,\mathbf{\Phi}(\bar{x}_n,z)\wedge y=\mathbf{g}(\bar{x}_n,z)\end{matrix}}{\mathbf{\Phi}(\bar{x}_n,z)\wedge y=\mathbf{g}(\bar{x}_n,z)}}{\dfrac{\mathbf{\Psi}(\bar{x}_n,y)}{\forall\bar{x}_n\,.\,\mathbf{P}\supset\exists y\,.\,\mathbf{\Psi}(\bar{x}_n,y)}}{\scriptstyle \supset\mathrm{E}}$$

Applying finite differencing to depth-first search

We now show how the transformation can be applied to Proof 5.34. Matching this proof against Proof schemas 5.45 through 5.47 yields the following instantiations:

- $\mathbf{g} : \lambda V . \lambda E . \lambda U . \lambda R . \lambda r . R \cup r$
- From the first base case:
  - $\bar{\mathbf{t}}_{1n} : V, E, \{\}, R$
  - $\mathbf{f}_1 : \lambda V . \lambda E . \lambda U . \lambda R . \{\}$
- From the second base case:
  - $\bar{\mathbf{t}}_{2n} : V, E, U, \{\}$
  - $\mathbf{f}_2 : \lambda V . \lambda E . \lambda U . \lambda R . \{\}$
- From the first inductive case:
  - $\bar{p}_{3j} : V, E, U, R, r$
  - $\bar{\mathbf{d}}_{3n} : V, E, U, R$
  - $\bar{\mathbf{t}}_{3n} : V, E, U, r \uplus R$
  - $\mathbf{f}_3 : \lambda V . \lambda E . \lambda U . \lambda R . \lambda r . \lambda y . y$
- From the second inductive case:
  - $\bar{p}_{4j} : V, E, U, R, r, s, S$
  - $\bar{\mathbf{d}}_{4n} : V, E, U \setminus \{s\}, (r \uplus R) \cup \{s\}$
  - $\bar{\mathbf{t}}_{4n} : V, E, U, r \uplus R$
  - $\mathbf{f}_4 : \lambda V . \lambda E . \lambda U . \lambda R . \lambda r . \lambda s . \lambda S . \lambda y . \{s\} \cup y$

With these instantiations we are in a position to examine the proof obligations (the equations) for the four cases of the induction.

1. $\mathcal{L}_1$ must prove
$$\mathbf{h}_1(V, E, U, R) = R \cup \{\}$$
for some $\mathbf{h}_1$; the obvious choice is $\lambda V . \lambda E . \lambda U . \lambda R . R$, so the proof obligation becomes
$$\forall R . R = R \cup \{\}$$
which is trivial to prove.

2. $\mathcal{L}_2$ must prove
$$\mathbf{h}_2(V, E, U, R) = \{\} \cup \{\}$$
so we choose the constant function yielding $\{\}$ for $\mathbf{h}_2$, and again have a trivial proof obligation of $\{\} = \{\} \cup \{\}$.

3. For $\mathcal{L}_3$ the user must find some $\mathbf{h}_3$ so that for any z
$$\mathbf{h}_3(V, E, U, R, r, R \cup z) = (r \uplus R) \cup z$$
Choosing $\lambda V . \lambda E . \lambda U . \lambda R . \lambda r . \lambda y . r \uplus y$ for $\mathbf{h}_3$ leads to the proof obligation
$$r \uplus (R \cup z) = (r \uplus R) \cup z$$
which follows easily from the associativity of set union.

4. For $\mathcal{L}_4$ the user must find some $\mathbf{h}_4$ so that for any z

$$\mathbf{h}_4(V, E, U, R, r, s, S, ((r \uplus R) \cup \{s\}) \cup z) = (r \uplus R) \cup (\{s\} \cup z)$$

Choosing $\lambda V . \lambda E . \lambda U . \lambda R . \lambda r . \lambda s . \lambda S . \lambda y . y$ for $\mathbf{h}_4$ leads to the proof obligation

$$((r \uplus R) \cup \{s\}) \cup z = (r \uplus R) \cup (\{s\} \cup z)$$

which again follows easily from the associativity of set union.

The result of the transformation is the following proof of Lemma 5.42:

Proof 5.51 We prove the following lemma by induction on the pair of sets (U, R):

Lemma 5.52

$$\forall \langle V, E \rangle . \forall U . \forall R . U, R \subseteq V \supset \exists W . \exists X . X = R \cup \widehat{E_U^+}(R) \wedge W = R \cup X$$

Case $U = \{\}$; then let $X = \{\}$ and $W = R$.
Case $R = \{\}$; then let $X = W = \{\}$.
Case $U \neq \{\}$, $R = r \uplus R'$. There are two cases:

1. $\widehat{E_U}(r) = \{\}$. Then apply the induction hypothesis to (U, R') to obtain $X' = \widehat{E_U}^+(R')$ and $W' = R' \cup X'$.

   By the reasoning of Proof 5.34 $\widehat{E_U}^+(R') = \widehat{E_U}^+(r \uplus R')$. So let $X = W'$. Thus $W' = R' \cup \widehat{E_U}^+(r \uplus R')$. Then by the associativity of set union $\{r\} \cup W' = (r \uplus R') \cup \widehat{E_U}^+(r \uplus R')$. So let $W = \{r\} \cup W'$.

2. $\widehat{E_U}(r) = s \uplus S$. Write U' for $U \setminus \{s\}$ and apply the induction hypothesis to $(U', R \cup \{s\})$ to obtain $X' = (\widehat{E_{U'}}^+)(R \cup \{s\})$ and $W' = (R \cup \{s\}) \cup W'$. By the reasoning of Proof 5.34 $\{s\} \cup \widehat{E_{U'}}^+(R \cup \{s\}) = \widehat{E_U}^+(R)$. So let $X = \{s\} \cup X'$. Now $W' = (R \cup \{s\}) \cup (\widehat{E_{U'}}^+)(R \cup \{s\}) = R \cup (\{s\} \cup (\widehat{E_{U'}}^+)(R \cup \{s\}))$ by associativity. Then $W' = R \cup \widehat{E_U}^+(R)$, so let $W = W'$.

Lemma 5.42 follows trivially. □

Once the proof is further transformed by inserting double negations, the following program can be extracted. It traverses the graph in depth-first order, adding nodes to the result as they are removed from the root set R. This corresponds to a post-order listing of a depth-first spanning forest with roots in R.

Program 5.53

```
fun rtclos' (G as (graph(V,E))) R =
let fun rtclos_1' G empty R = R
      | rtclos_1' G U empty = empty
      | rtclos_1' (G as graph(V,E)) U (R as (r++R')) =
          case ((mapp(r,G)) && U) of
            empty => r ++ (rtclos_1' G U R')
          | (s++S) =>
             (rtclos_1' G (U -- s) (s++R))
in (rtclos_1' G V R) end
```

5.3 Discussion

The two case studies of this chapter demonstrate that simple proof transformations expressible as derived rules of the object logic can be used to obtain results comparable to those of program transformation while maintaining a formal basis (the object proof) for an explanation of the program. The capability of proof transformation to alter the functionality of the extracted program, already seen in the example of Chapter 4, appeared again in the case of breadth-first search. However, the significance of this capability remains to be seen. It may be more important that this methodology integrates the flexibility of a transformational approach with the formal connection between a program and its specification provided by the proofs-as-programs methodology.

The program development of the depth-first search case study can be duplicated at the program level by fold-unfold program transformations combined with Sasaki's dead code elimination technique [Sas86]. Dead code elimination would produce the transitive closure function Program 5.40 from Program 5.37. The developer could then define reflexive transitive closure in terms of set union and transitive closure. Fold-unfold transformations would eventually yield our final program. But a proof that this meets the specification of reflexive transitive closure is external to this process, while such a proof is integral to the proof transformation approach.

The proofs produced by the transformations of the case studies could of course be developed directly. The end product of the depth-first search development, in particular, is no more complicated or difficult to understand than the initial proof. Part of the point of the example is to show how transformations can be used to avoid re-doing a proof from the beginning when the specification changes slightly. The transformation isolates proof obligations so that a user does not need to repeat the theorem-proving tasks already done. Where the end product of the development is more complex than the "natural proof," a record of the transformations used to obtain it may be useful as a formal basis for documentation of the extracted program. The proof for the tail-recursive form of breadth-first search is a small example of this: to an uninformed reader the presence of the accumulator parameter requires explanation. A large, highly optimized proof will likely contain many such obscure points. Keeping such a record would amount to lifting the *design record* advocated in [SS83] from the domain of programs to the domain of proofs.

Chapter 6

Conclusion

This chapter summarizes the thesis and discusses some directions for further research.

6.1 Summary

This work has explored a transformational extension to the proofs-as-programs methodology. We have focused on a declarative formulation of proof transformations and a partially verified implementation based on a logical framework and higher-order logic programming. Our case studies show that known program transformations can be translated into the domain of proofs in the form of derived rules of the object logic. In the case of tail recursion introduction, we demonstrated the effect noted by Goad for the case of specialization: the increased power of such a proof transformation over its corresponding purely syntactic program transformation.

We gave an implementation of the core of the support for programming as theorem-proving: a constructive logic, the syntax and semantics of a small functional programming language, and the extraction of programs from proofs. There are far more advanced systems capable of supporting programming by theorem proving, and they are increasing rapidly in sophistication. But these systems are committed to a particular logic and programming language and provide limited support for verified metaprogramming. The framework approach allows for rapid experimentation with various object logics, programming languages, notions of program extraction, and metaprograms.

The implementation is concise and was easy to develop largely because of an intensive use of higher-order abstract syntax. This was possible because of the choice of logic, programming language, and the kind of proof transformations to be treated. However, it is important to stress that the use of a logical framework is not restricted to cases where higher-order abstract syntax is an appropriate technique.

We have shown how our choice of implementation strategy supports verified metaprogramming. Proofs of correctness properties of program extraction were partially internalized as Elf programs in such a way that type checking of the programs amounts to the checking of the constructive parts of the proofs. We encoded proof transformations as rewrite rules so that their type-correctness is equivalent to a proof that they are derived rules of the object logic. This style of encoding gives a strong guarantee of correctness, not only for the result of a particular application of the transformation, but for the transformation itself.

While we have not given a formal method for the translation of known program transformations to the domain of proofs, the case studies demonstrate that such translation is feasible, and that the result is likely to be expressible as a derived logical rule. We gave rules in this form for transformations to introduce tail-recursive structure, to perform finite differencing, and to change the basis of a recursive definition, and applied them to several small programming problems on lists, trees, and directed graphs. Our study of tail-recursion showed that proof transformation resulted in a useful change to the functional behavior of the extracted program, which would not be obtained by syntactic program transformations that preserve functionality. Although Goad demonstrated this effect for specialization, we do not know of any other examples in the literature.

6.2 Assessment

The implementation and case studies are evidence that metaprograms for treating proof systems can be easily implemented in a declarative way that lends itself to verification, and effectively applied to useful tasks in program development.

Although we used a very simple object logic, in principle the approach is independent of this choice. While encodings of richer logics or special-purpose logics are not necessarily trivial to achieve, there is a wide range of choice even with the particular framework we used. Given the current interest in variations on logics and programming languages for proofs-as-programs (e.g. [Mur90], [PW90], [Raf93]), the implementation methodology demonstrated here can be a valuable aid to research.

The methodology we used can work with a wide variety of theorem-proving tools. The only requirement is that proof objects can be generated by the proof process. Thus it could be used in conjunction with interactive proof assistants such as Coq or Nuprl, or with the addition of proof objects, automated deduction systems like the Boyer-Moore prover [BM79] or provers that are components of larger systems like the software development system KIDS [Smi91].

The potential of our approach is not limited to applications to program development by the proofs-as-programs methodology, or to programming in the small, or even to programming. It can be used wherever the adaptation and reuse of formal deductions is of interest. With further research, the use of proof transformations could have an impact on methodologies and support tools for a variety of tasks where formal proof is important. This impact will come primarily from the fact that proof transformation offers support for the modification and reuse of formal proofs that provide a basis for verification and documentation, reducing the cost of these activities. The framework-based implementation strategy used in this thesis provides the flexibility and ease of programming necessary to support experimentation in new areas of application. The variety of possible settings in which it will be interesting to explore the use of proof transformation is an argument in favor of our implementation strategy, at least as a basis for research. Its flexibility and independence of the choice of logic and theorem-proving tools can support quite different lines of research, yet allow sharing of techniques between them.

6.3 Future work

The implementation described here could be improved and extended in a number of ways. The work of Felty [Fel89] on the specification and implementation of theorem provers in the closely related setting of λProlog, combined with our work, would provide a basis for a complete implementation in Elf of support for programming by theorem proving. The tacticals she developed for theorem provers in λProlog could be adapted to the problem of high-level control of the proof transformation process.

Planned extensions to the Elf language promise improved support for the structuring of object languages, theories, and metaprograms. The module system of [HP99] would greatly ease the task of creating and maintaining a scaled-up implementation, by providing name-space management and explicit representation of the relations between deductive systems in the form of parametrized signatures and realizors. Intersection types as proposed in [Pfe92b] could be used to decrease the duplication of code in the encoding of many-sorted first-order logic.

While the expression of a proof transformation as a derived logical rule has advantages for expressiveness and automatic verification, it is too specific. For instance, tail recursion introduction had to be encoded separately for different induction principles, and the encoding is also sensitive to different case analysis structures in the input proof. The generality of the transformations can be increased by coding them as logic programs that handle a class of proof structures, but at the cost of conciseness and declarative form. A general treatment of induction for the object logic along the lines of Hayashi's conditional inductive generations [Hay90] or the inductively defined types of Coq [PM93] might be one way to attack this problem. It would also be interesting to try to formulate transformations as signatures parametrized to a particular induction principle using the proposed Elf module system.

Our formulation of proof transformations does not completely specify the associated theorem-proving obligations. In most cases the transformation represents an obligation as a pattern which is only partially instantiated by unification with the input proof. Peter Madden has shown in his thesis that similar problems for the tupling transformation can be solved by heuristics and analysis. This work could certainly be extended to our implementation framework.

Further experimentation with case studies will be required to understand the potential value of proof transformation for the program development process. It would be valuable to find a characterization of circumstances in which proof transformation can do more than syntactic program transformation. The tail-recursion examples that we studied produced a change in the functionality of the extracted program. But as we noted, this would not occur if the problems were specified on the domain of sets rather than lists, because the corresponding functions on sets are equal. The specification in terms of lists may be thought of as arising from a commitment to a particular concrete data type; from this perspective the proof transformation is a tool for recovering from a premature commitment, a common problem in software maintenance.

Further case studies could also reveal to what extent proof transformations can syntactically capture heuristics and semantic side conditions for program transformations. Since a proof describes formally how a program implements its specification, it may contain more clues to how the program can be transformed than the program itself does. As we have seen, proof transformations necessarily capture proof obligations for a program transformation. Perhaps existing program transformation approaches could be enriched with a proof-transformational formulation that would provide a uniform language for expressing some heuristic knowledge and all of the semantic correctness criteria.

These case studies should ideally explore hard program development problems like those treated by the state-of-the-art program transformation and synthesis research. This will require a richer object logic than was used in this work, such as Nuprl or Constructions, as well as good proof development tools.

One way to attack such examples would be to do the work entirely within a proof development system like Coq, implementing proof transformations in the ML metalanguage. Although the metaprograms would be less easily verified and less declarative in style than the encodings developed in this thesis, this approach could lead to insights into how such proof development systems can support the reuse and adaptation of proofs in general, not only proofs of programs. A variation on this approach would be to use the chosen system for theorem-proving but interface it to a framework like Elf where proof transformations could be more easily represented and verified.

An alternative line of research is to view proof transformation as an enrichment of program transformation, and to build more directly on research in that area. For example, many features of Smith's KIDS system [Smi91] could in principle be recast as proof constructions and proof transformations. Theorem-proving in this system is accomplished by goal-directed inference, driven by the needs of the programming problem at hand. This approach to proof fits well with the way proof transformations impose proof obligations. Directed inference may be well-suited to the problem noted above of fully instantiating proof obligations, as well as that of fulfilling those obligations. Some of the optimizations of KIDS have already been translated to the domain of proofs, for example specialization (by Goad) and finite differencing (in this thesis). The treatment of abstract data types suggested by Pfenning [Pfe90] could probably be used as a proof-transformational basis for the data type refinement feature of KIDS. Implementing the major features of a system like KIDS in a proofs-as-programs setting augmented with proof transformations would be a hard but rewarding task. Because KIDS contains a great deal of programming knowledge and high-level operators, such an implementation would provide significant help with case studies of the application of the proofs-as-programs methodology to difficult programming problems. Proof objects could provide a uniform framework for the representation of the information that is exploited at each development step. This is potentially valuable for both the correctness of the implementation of the transformation system and the documentation of the program development process.

Appendix A

Elf encodings from Chapter 2

All Elf code in this thesis is accessible (at the time of writing) by anonymous ftp from Carnegie Mellon University. To obtain the code:

```
% ftp ftp.cs.cmu.edu
Name: anonymous
Password: (your e-mail address)
ftp> cd /afs/cs/user/apa/ftp
ftp> type binary
ftp> get thesis-elf.tar.Z
ftp> bye

% uncompress thesis-elf.tar.Z
% tar -xvf thesis-elf.tar
% rm *.tar
```

(ftp.cs.cmu.edu has internet address 128.2.206.173)

This will create a directory proof-trans/ with the Elf code and examples in various subdirectories. For the Elf implementation itself see the file `/afs/cs/user/fp/public/README`.

A.1 First-order logic and arithmetic

```
%%% First-order constructive logic.

%%% Static

%%% Propositions.
%%% Syntactic categories.
o : type.                             %name o A B C
i : type.                             %name i T1 T2 T3

%%% Syntax.
```

```
true : o.
false : o.
not : o -> o.
and : o -> o -> o.
or : o -> o -> o.
implies : o -> o -> o.
forall : (i -> o) -> o.
exists : (i -> o) -> o.

%%% Proofs
|- : o -> type.                              %name |- P Q P1 P2 P3 Q1 Q2 Q3

truei : |- true.

falsee : {C:o} |- false -> |- C.

andi : |- A -> |- B -> |- (and A B).
andel : |- (and A B) -> |- A.
ander : |- (and A B) -> |- B.

oril : {B:o} |- A -> |- (or A B).
orir : {A:o} |- B -> |- (or A B).
ore : (|- A -> |- C) -> (|- B -> |- C) -> |- (or A B) -> |- C.

impliesi : (|- A -> |- B) -> |- (implies A B).
impliese : |- (implies A B) -> |- A -> |- B.

noti : (|- A -> |- false) -> |- (not A).
note : |- (not A) ->  |- A  -> |- false.

foralli : ({x:i} |- (A x)) -> |- (forall A).
foralle : {T:i} |- (forall A) -> |- (A T).

existsi : {A:i -> o} {T:i} |- (A T) -> |- (exists A).
existse : ({x:i} |- (A x) -> |- C) -> |- (exists A) -> |- C.

%%% Interpret "i" of FOL as natural numbers.

%%% Static.

zero : i.
succ : i -> i.

%% Equality
eq :  i -> i -> o.

eq_refl : {X:i} |- (eq X X).
eq_sym : |- (eq X Y) -> |- (eq Y X).
eq_trans : |- (eq X Y) -> |- (eq Y Z) -> |- (eq X Z).
eq_subst : {F:i -> i} |- (eq X Y) -> |- (eq (F X) (F Y)).
```

```
ax_zero : {X:i} |- (not (eq (succ X) zero)).
ax_succ : {X:i} {Y:i} |- (eq (succ X) (succ Y)) -> |- (eq X Y).

%% Induction

ind : {A:i -> o} |- (A zero) -> ({x:i} |- (A x) -> |- (A (succ x)))
        -> |- (forall A).
```

A.2 Functional programming language

```
%%% Abstract syntax of terms and types.

%%% Static.

tp : type.                                              %name tp Tp Tp' Tp''
term : type.                                            %name term M N M' N' M'' N''

arrow : tp -> tp -> tp.

app : term -> term -> term.
lam : (term -> term) -> term.
fix : (term -> term) -> term.

unit : tp.
unity : term.

nat : tp.
0 : term.
s : term -> term.                  %prefix 100 s
nat_ind : term -> (term -> term -> term) -> term.

* : tp -> tp -> tp.
pair : term -> term -> term.
fst : term -> term.
snd : term -> term.
spread : term -> (term -> term -> term) -> term.

| : tp -> tp -> tp.
inl : term -> term.
inr : term -> term.
decide : term -> (term -> term) -> (term -> term) -> term.

void : tp.
any : term -> term.
neg : term.

atom : tp.
axiom : term.
```

```
%%% Call-by-value natural semantics.

eval : term -> term -> type.

ev_0 : eval 0 0.
ev_s : eval (s M) (s V) <- eval M V.

ev_pair : eval (pair M N) (pair V V') <- eval M V <- eval N V'.
ev_spread : eval (spread M N) V
            <- eval M (pair V1 V2)
            <- eval (N V1 V2) V.
ev_fst : eval (fst M) V1 <- eval M (pair V1 V2).
ev_snd : eval (snd M) V2 <- eval M (pair V1 V2).

ev_inl : eval (inl M) (inl V) <- eval M V.
ev_inr : eval (inr M) (inr V) <- eval M V.
ev_dec_l : eval (decide M Nl Nr) V
             <- eval M (inl V')
             <- eval (Nl V') V.
ev_dec_r : eval (decide M Nl Nr) V
             <- eval M (inr V')
             <- eval (Nr V') V.

ev_lam : eval (lam M) (lam M).
ev_app_lam : eval (app M N) V
             <- eval M (lam M')
             <- eval N V1
             <- eval (M' V1) V.

ev_pr : eval (nat_ind M0 Ms) (nat_ind M0 Ms).
ev_pr_z : eval (app M N) V
          <- eval M (nat_ind M0 Ms)
          <- eval N 0
          <- eval M0 V.
ev_pr_s : eval (app M N) V
          <- eval M (nat_ind M0 Ms)
          <- eval N (s N')
          <- eval (Ms N' (app (nat_ind M0 Ms) N')) V.

ev_unity : eval unity unity.
ev_axiom : eval axiom axiom.
ev_neg : eval neg neg.


%%% Type inference.


of : term -> tp -> type.
```

```
tp_0 : of 0 nat.
tp_s : of (s M) nat <- of M nat.

tp_pair : of (pair M N) (* A B) <- of M A <- of N B.
tp_fst :  of (fst M) A <- of M (* A B).
tp_snd :  of (snd M) B <- of M (* A B).
tp_spread : of (spread Mpr N) C
            <- of Mpr (* A B)
            <- {x} of x A -> {y} of y B -> of (N x y) C.

tp_inl :  {B} of (inl M) (| A B) <- of M A.
tp_inr :  {A} of (inr M) (| A B) <- of M B.
tp_decide : of (decide M Nl Nr) C
            <- of M (| A B)
            <- ({x:term} of x A -> of (Nl x) C)
            <- ({x:term} of x B -> of (Nr x) C).

tp_lam : of (lam M) (arrow A B)
            <- {x:term} of x A -> of (M x) B.
tp_app : of (app M N) B <- of M (arrow A B) <- of N A.

tp_prec : of (nat_ind Mz Ms) (arrow nat A)
          <- of Mz A
          <- ({x} of x nat -> {y} of y A -> of (Ms x y) A).

tp_unity : of unity unit.
tp_axiom : of axiom atom.
tp_any : {A} of (any M) A <- of M void.
tp_neg : {A} of neg (arrow A void).
```

A.3 Program and type extraction

```
%%% Extraction from individual terms of arithmetic.

%%% DYNAMIC

extract_tm : i -> term -> type.

ex_zero : extract_tm zero 0.
ex_succ : extract_tm (succ X) (s M) <- extract_tm X M.



%%% Extraction of fpl functions from
%%% intuitionistic proofs.

%%% Dynamic.
```

```
extract : |- A -> term -> type.

ex_truei : extract truei unity.

ex_falsee : extract (falsee C P) (any M) <- extract P M.

ex_andi : extract (andi P1 P2) (pair M1 M2)
            <- extract P1 M1 <- extract P2 M2.
ex_andel : extract (andel P) (fst M) <- extract P M.
ex_ander : extract (ander P) (snd M) <- extract P M.

ex_oril : extract (oril B P) (inl M) <- extract P M.
ex_orir : extract (orir A P) (inr M) <- extract P M.

ex_ore : extract (ore P1 P2 Q) (decide M Nl Nr)
          <- extract Q M
          <- ({x:|- A} {v:term} extract x v -> extract (P1 x) (Nl v))
          <- ({x:|- B} {v:term} extract x v -> extract (P2 x) (Nr v)).

ex_impliesi : extract (impliesi P) (lam M)
               <- {x:|- A} {v:term} extract x v -> extract (P x) (M v).

ex_impliese : extract (impliese P1 P2) (app M1 M2)
                <- extract P1 M1 <- extract P2 M2.

ex_noti : extract (noti P) (lam M)
            <- {x:|- A} {v:term} (extract x v -> extract (P x) (M v)).
ex_note : extract (note P1 P2) (app M1 M2)
            <- extract P1 M1 <- extract P2 M2.

%%% Extraction for arithmetic

%%% Dynamic.

ex_foralli : extract (foralli P) (lam M)
                <- {x:i} {v:term} extract_tm x v -> extract (P x) (M v).

ex_foralle : extract (foralle T P) (app M N)
                <- extract P M <- extract_tm T N.

ex_existsi : extract (existsi A T P) (pair N M)
                <- extract_tm T N <- extract P M.

ex_existse : extract (existse P Q) (spread N M)
                <- ({x:i} {x':term} extract_tm x x'
                      -> {p:|- (A x)} {p':term} extract p p'
                            -> extract (P x p) (M x' p'))
                <- extract Q N.

ex_refl : extract (eq_refl X) axiom.
ex_sym : extract (eq_sym P) axiom.
```

```
ex_trans : extract (eq_trans P1 P2) axiom.
ex_subst : extract (eq_subst F P) axiom.
ex_ax_zero : extract (ax_zero X) neg.
ex_ax_succ : extract (ax_succ X Y P) axiom.

ex_ind : extract (ind A Pz Ps) (nat_ind Nz Ns)
         <- extract Pz Nz
         <- {x:i} {v:term} extract_tm x v
            -> {p:|- (A x)} {w:term} extract p w
            -> extract (Ps x p) (Ns v w).


%%% Extraction of fpl types.

%%% Dynamic.

extract_tp : o -> tp -> type.

ext_true : extract_tp true unit.

ext_false : extract_tp false void.

ext_not : extract_tp (not A) (arrow Tp void)
          <- extract_tp A Tp.

ext_and : extract_tp (and A B) (* Tp1 Tp2)
          <- extract_tp A Tp1
           <- extract_tp B Tp2.

ext_or : extract_tp (or A B) (| Tp1 Tp2)
          <- extract_tp A Tp1
           <- extract_tp B Tp2.

ext_implies : extract_tp (implies A B) (arrow Tp1 Tp2)
          <- extract_tp A Tp1
           <- extract_tp B Tp2.

%%% Extraction of fpl types.

%%% Dynamic.

ext_forall : extract_tp (forall A) (arrow nat Tp)
             <- {x:i} extract_tp (A x) Tp.

ext_exists : extract_tp (exists A) (* nat Tp)
             <- {x:i} extract_tp (A x) Tp.

ext_eq : extract_tp (eq X Y) atom.
```

A.4 Extraction/simplification

```
%%% Extraction/simplification of fpl terms from intuitionistic proofs.

extract_simp : |- A  -> term -> type.

%% Uninformative proofs
exs_n: extract_simp (P: |- (not A)) neg.
exs_f: extract_simp (P: |- false) (app neg unity).
exs_un: extract_simp (P: |- A) unity <- uninf A.

%% CONJUNCTION
%% Don't build (pair M N:B) if B is uninformative.
exs_andi1 : extract_simp (andi (P: |- A) (Q: |- B)) M
            <- inf A  <- extract_simp P M <- uninf B.

%% Don't build (pair M:A N) if A is uninformative.
exs_andi2 : extract_simp (andi (P: |- A) (Q: |- B)) M
            <- uninf A <- inf B <- extract_simp Q M.

exs_andi3 : extract_simp (andi (P: |- A) (Q: |- B)) (pair M N)
            <- inf A <- extract_simp P M
            <- inf B <- extract_simp Q N.

%% Don't build (fst M:(* A B)) if B is uninformative.
exs_andel1 : extract_simp (andel (P : |- (and A B))) M
             <- inf A <- uninf B <- extract_simp P M.

exs_andel2 : extract_simp (andel (P : |- (and A B))) (fst M)
             <- inf A <- inf B
             <- extract_simp P M.

%% Don't build (snd M:(* A B)) if A is uninformative.
exs_ander1 : extract_simp (ander (P : |- (and A B))) M
        <- uninf A <- inf B <- extract_simp P M.

exs_ander2 : extract_simp (ander (P : |- (and A B))) (snd M)
             <- inf A <- inf B
             <- extract_simp P M.

%% DISJUNCTION
exs_oril : extract_simp (oril B P) (inl M)
        <- extract_simp P M.

exs_orir : extract_simp (orir A P) (inr M)
        <- extract_simp P M.

exs_ore : extract_simp (ore P1 P2 Q) (decide M Nl Nr)
        <- extract_simp Q M
        <- ({P:|- A} {p:term} extract_simp P p -> extract_simp (P1 P) (Nl p))
        <- ({P:|- B} {p:term} extract_simp P p -> extract_simp (P2 P) (Nr p)).
```

```
%% IMPLICATION
%% Don't build (lam M) if A is uninformative.
exs_impli1 : extract_simp (impliesi (P: |- A -> |- B)) M
             <- uninf A <- inf B
             <- {p: |- A} extract_simp (P p) M.

exs_impli2 : extract_simp (impliesi (P: |- A -> |- B)) (lam M)
             <- inf A <- inf B
             <- {Q: |- A} {q:term} (extract_simp Q q -> extract_simp (P Q) (M q)).

%% Don't build (app M N:A) if A is uninformative.
exs_imple1 : extract_simp (impliese (P: |- (implies A B)) Q) M
             <- uninf A <- inf B <- extract_simp P M.

exs_imple2 : extract_simp (impliese (P: |- (implies A B)) Q) (app M N)
             <- inf A <- inf B
             <- extract_simp P M <- extract_simp Q N.

%% NEGATION
%% (noti P) will always be caught by the rule "exs_n" since the conclusion is "(not A)".
%% (note P) will always be caught by the rule "exs_f" since the conclusion is "false".

%% ABSURDITY
exs_falsee : extract_simp (falsee C P) (any M)
             <- extract_simp P M.
%%% ASSERTION
exs_isay : extract_simp (isay A) unity.

%%% Extraction with removal of uninformative parts.
%%% Requires pap/extract-simp.elf

%%% DYNAMIC

%% UNIVERSAL QUANTIFICATION
exs_foralli : extract_simp ((foralli P):|- (forall A)) (lam M)
              <- inf (forall A)
              <- {X:i} {x:term} (extract_tm X x -> extract_simp (P X) (M x)).

exs_foralle : extract_simp (foralle T (P: |- (forall A))) (app M N)
              <- inf (forall A)
              <- extract_simp P M <- extract_tm T N.

%% EXISTENTIAL QUANTIFICATION
%% Don't build (pair M N:A) if A is uninformative.
exs_existsi1 : extract_simp (existsi A T _) M
               <- ({x:i} uninf (A x)) <- extract_tm T M.
exs_existsi2 : extract_simp (existsi A T P) (pair M N)
               <- ({x:i} inf (A x))
               <- extract_tm T M <- extract_simp P N.
```

```
%% We don't have (pair M N:A) if A is uninformative.
exs_existse1 : extract_simp (existse P (Q: |- (exists A))) (app (lam M) N)
                <- ({X:i} uninf (A X))
                <- ({X:i} {x:term} extract_tm X x
                    -> {p: |- (A X)} extract_simp (P X p) (M x))
                <- extract_simp Q N.

exs_existse2 : extract_simp (existse P_min P_maj) (spread N M)
                <- ({X:i} inf (A X))
                <- ({X:i} {x:term} extract_tm X x
                    -> {P:|- (A X)} {p:term} extract_simp P p
                    -> extract_simp (P_min X P) (M x p))
                <- extract_simp P_maj N.

%% INDUCTION
exs_ind : extract_simp (ind A Pz Ps) (nat_ind Nz Ns)
           <- ({X} inf (A X))
           <- extract_simp Pz Nz
           <- {X:i} {x:term} extract_tm X x
               -> {P:|- (A X)} {p:term} extract_simp P p
               -> extract_simp (Ps X P) (Ns x p).


%%% Extract simplified type

%%% Dynamic.

ex_tp_s : o -> tp -> type.

exts_h: ex_tp_s A unit <- uninf A.
exts_n: ex_tp_s (not A) (arrow T void) <- ex_tp_s A T.
exts_false : ex_tp_s false void.

exts_andl : ex_tp_s (and A B) T
             <- inf A <- ex_tp_s A T
             <- uninf B.
exts_andr : ex_tp_s (and A B) T
             <- uninf A
             <- inf B <- ex_tp_s B T.
exts_and : ex_tp_s (and A B) (* T1 T2)
            <- inf A <- ex_tp_s A T1
            <- inf B <- ex_tp_s B T2.

exts_or : ex_tp_s (or A B) (| T1 T2)
           <- ex_tp_s A T1
           <- ex_tp_s B T2.

exts_impr : ex_tp_s (implies A B) T
             <- uninf A
             <- inf B <- ex_tp_s B T.
exts_imp : ex_tp_s (implies A B) (arrow T1 T2)
```

```
          <- inf A <- ex_tp_s A T1
          <- inf B <- ex_tp_s B T2.

exts_all : ex_tp_s (forall A) (arrow nat T)
          <- ({x:i} inf (A x)) <- {x:i} ex_tp_s (A x) T.

exts_exl : ex_tp_s (exists A) nat
          <- ({x:i} uninf (A x)).

exts_ex : ex_tp_s (exists A) (* nat T)
         <- ({x:i} inf (A x))
         <- ({x:i} ex_tp_s (A x) T).
```

Appendix B

Adequacy of encodings

This appendix discusses the faithfulness of the encodings of the object languages and inference systems of Chapter 2. Following Harper et al. [HHP93], we describe minimal correctness properties, called *adequacy* theorems, for the encodings. Proofs of adequacy tend to be rather stereotyped and filled with unenlightening detail, so we omit them.

Adequacy in general consists of two properties: the existence of a bijective representation function from the expressions of a language to be represented to terms of the formalized representation, and the *compositionality* of the representation function. The bijection ensures that every language expression has a unique representation as a canonical term of the representing type, and that every such canonical term represents a language expression. Loosely speaking, a representation function is compositional when it commutes with substitution. Thus to state an adequacy theorem in full it is necessary to define a representation function, possibly parameterized to a set of variables if open terms must be treated, and to define substitution in both the language to be represented and the representing formalism. The key idea in such theorems is to identify the variables of the represented language with the variables of the formalism. This identification reflects the basic principle of representations based on higher-order abstract syntax.

The encodings of first-order logic and natural deduction we used closely follow those of Harper et al. Although their encoding is for classical logic, this does not materially affect the statements and proofs of correctness (the major change is to omit the classical absurdity rule from the natural deduction inference system). Rather than restate the adequacy theorems for these we refer the interested reader to [HHP93].

We assume as background the definition of canonical forms for LF terms and the proof of their existence.

B.1 Adequacy for syntax of program expressions and types

Adequacy for the functional programming language and its type system is easy to state. We omit the definition of substitution as it is the usual one. The definitions of representation functions use typewriter font for LF/Elf term constructors (e.g., `pair`). An expression of the form $\lambda x \,.\, e$ represents an LF/Elf term of functional type.

We repeat the syntax of the function programming language used for extraction:

e ::=	x |	*Variables*
	0 | **s**(e) |	*Natural numbers*
	$\langle\, e_1, e_2 \,\rangle$ | **fst**(e) | **snd**(e) | **spread**(e_1; $x,y\,.\,e_2$) |	*Pairs*
	inl(e) | **inr**(e) | **decide**(e_1; $x\,.\,e_2$; $x\,.\,e_3$) |	*Disjoint union*
	lam $x\,.\,e$ | **nat_ind**(e_1; $x,y\,.\,e_2$) | **app**(e_1,e_2) |	*Functions*
	() |	*Unity*
	any(e) | **neg**	*Error*
	axiom	*Self-realizors*

The Elf signature for the representation of programming language expressions and types, which we denote by Σ_{PL}:

```
tp : type.
term : type.

arrow : tp -> tp -> tp.

app : term -> term -> term.
lam : (term -> term) -> term.

unit : tp.
unity : term.

nat : tp.
0 : term.
s : term -> term.
nat_ind : term -> (term -> term -> term) -> term.

* : tp -> tp -> tp.
pair : term -> term -> term.
fst : term -> term.
snd : term -> term.
spread : term -> (term -> term -> term) -> term.

| : tp -> tp -> tp.
inl : term -> term.
inr : term -> term.
decide : term -> (term -> term) -> (term -> term) -> term.

void : tp.
any : term -> term.
neg : term.

atom : tp.
```

```
axiom : term.
```

Definition B.1 *Given a set of variables $X = \{x_1, \ldots, x_n\}$, we define the corresponding Elf context Γ_X as $\{x_1 : \mathtt{term}, \ldots, x_n : \mathtt{term}\}$.*

Definition B.2 *Let X be a set of variables. The following defines the representation function ε_X for programming language expressions. The notation $\varepsilon_{X,x_1,\ldots,x_n}$ denotes the representation function for the set X extended by the addition of the variables $x_1, \ldots, x_n$.*

$\varepsilon_X(x) = x$, *if* $x \in X$
$\varepsilon_X(0) = \mathtt{0}$
$\varepsilon_X(\mathbf{s}(e)) = \mathtt{s}\ \varepsilon_X(e)$
$\varepsilon_X(\langle e_1, e_2 \rangle) = \mathtt{pair}\ \varepsilon_X(e_1)\ \varepsilon_X(e_2)$
$\varepsilon_X(\mathbf{fst}(e)) = \mathtt{fst}\ \varepsilon_X(e)$
$\varepsilon_X(\mathbf{snd}(e)) = \mathtt{snd}\ \varepsilon_X(e)$
$\varepsilon_X(\mathbf{spread}(e_1;\ x, y \,.\, e_2)) = \mathtt{spread}\ \varepsilon_X(e_1)\ (\lambda x : \mathtt{term}.\, \lambda y : \mathtt{term}.\, \varepsilon_{X,x,y}(e_2))$
$\varepsilon_X(\mathbf{inl}(e)) = \mathtt{inl}\ \varepsilon_X(e)$
$\varepsilon_X(\mathbf{inr}(e)) = \mathtt{inr}\ \varepsilon_X(e)$
$\varepsilon_X(\mathbf{decide}(e_1;\ x \,.\, e_2;\ x \,.\, e_3)) = \mathtt{decide}\ \varepsilon_X(e_1)\ (\lambda x : \mathtt{term}.\, \varepsilon_{X,x}(e_2))\ (\lambda x : \mathtt{term}.\, \varepsilon_{X,x}(e_3))$
$\varepsilon_X(\mathbf{lam}\, x \,.\, e) = \mathtt{lam}\ \lambda x : \mathtt{term}.\, \varepsilon_{X,x}(e)$
$\varepsilon_X(\mathbf{nat\_ind}(e_1;\ x, y \,.\, e_2)) = \mathtt{nat\_ind}\ \varepsilon_X(e_1)\ \lambda x : \mathtt{term}.\, \lambda y : \mathtt{term}.\, \varepsilon_{X,x,y}(e_2)$
$\varepsilon_X(\mathbf{app}(e_1, e_2)) = \mathtt{app}\ \varepsilon_X(e_1)\ \varepsilon_X(e_2)$
$\varepsilon_X(()) = \mathtt{unity}$
$\varepsilon_X(\mathbf{any}(e)) = \mathtt{any}\ \varepsilon_X(e)$
$\varepsilon_X(\mathbf{neg}) = \mathtt{neg}$
$\varepsilon_X(\mathbf{axiom}) = \mathtt{axiom}$

Theorem B.3 Adequacy for programming language syntax: *For any set of variables X, ε_X is a bijection between the expressions of the programming language with free variables in X and the canonical forms of type* `term` *in the signature Σ_{PL} and the context Γ_X. The encoding is compositional, i.e., for a program expression e with free variables in $X = \{x_1, \ldots, x_n\}$ and program expressions $e_1, \ldots, e_n$ with free variables in Y,*

$$\varepsilon_Y([e_1/x_1, \ldots, e_n/x_n]e) = [\varepsilon_Y(e_1)/x_1, \ldots, \varepsilon_Y(e_n)/x_n]\varepsilon_X(e)$$

Adequacy for programming language types is simpler to state since there are no type variables. The obvious definition of the representation function ε for types can be made without reference to a set of variables – $\varepsilon(\tau_1 \times \tau_2) = *\ \varepsilon(\tau_1)\ \varepsilon(\tau_2)$, etc. Compositionality does not apply, so adequacy amounts to:

Theorem B.4 Adequacy for programming language types: *ε is a bijection between the type expressions of the programming language and the canonical forms of type* `tp` *in the signature Σ_{PL}.*

B.2 Adequacy for programming language semantics

The correctness properties for evaluation and type assignment deductions can be stated in much the same way. Evaluation is particularly simple since only closed programming language expressions occur in evaluation deductions, and these deductions do not introduce any assumptions. Thus there is no notion of substitution for an evaluation deduction, and an adequacy theorem can be formulated as for the representation of programming language types.

Type assignment deductions introduce assumptions; thus to state adequacy it is necessary (again following Harper et al.) to slightly reformulate the system to be represented so that assumptions are labelled by variables. Then the substitution of a type assignment deduction $\mathcal{T} :: X \vdash e \in \tau$ for an assumption variable $\xi :: x \in \tau$ can be defined so that substitution preserves the validity of deductions. This entails simultaneously substituting x for e in the programming language expressions of the deduction, as well as combining sets of typing assumptions. The representation function ε is then defined in terms of the representation functions for programming language expressions and types. Since representation of programming language expressions is parameterized by a set of programming language variables, we define:

Definition B.5 *Given a set of typing assumptions* $X = \{\xi_1 :: x_1 \in \tau_1, \ldots, \xi_n :: x_n \in \tau_n\}$, *the projection to programming language variables* $p(X)$ *is* $\{x_1, \ldots, x_n\}$.

Figure 2.11 shows the Elf signature that represents the type assignment system of Figure 2.10. However, this form of the signature exploits the Elf term and type reconstruction facility to avoid showing implicit arguments that must be made explicit in order to formulate statements of adequacy. In practice these arguments are supplied by Elf; we show a full declaration for the type inference rule for λ-abstraction as an example:

```
tp_lam : {M: term -> term} {A: tp} {B: tp}
         of (lam M) (arrow A B)
            <- {x:term} of x A -> of (M x) B.
```

Here the argument `M` of the constant `lam`, and the types `A` and `B` are made explicit. The arguments of all other constants in the signature can be similarly treated. With the signature Σ_{PL} that defines the syntax, let the resulting signature be Σ_{TP}. The representation function can then be defined in terms of Σ_{TP}. We show the definition for functional abstraction as an example:

$$\varepsilon\left(\dfrac{\begin{array}{c}\mathcal{T}\\ X, x \in \tau_1 \vdash e \in \tau_2\end{array}}{X \vdash \mathbf{lam}\, x\,.\, e \in \tau_1 \Rightarrow \tau_2}\,\textsf{tp-lam}\right) =$$

$$\texttt{tp\_lam}\ (\lambda x : \texttt{term}\,.\,\varepsilon_{p(X),x}(e))\ \varepsilon(\tau_1)\ \varepsilon(\tau_2)\ (\lambda x : \texttt{term}\,.\,\lambda\xi : \texttt{of}\ \ x\ \varepsilon(\tau_1)\,.\,\varepsilon(\mathcal{T}))$$

Definition B.6 *Given a set of typing assumptions* $X = \{\xi_1 :: x_1 \in \tau_1, \ldots, \xi_n :: x_n \in \tau_n\}$, *we define the corresponding Elf context* Γ_X *as* $\{x_1 : \texttt{term}, \ldots, x_n : \texttt{term}, \xi_1 : \texttt{of}\ x_1\ \varepsilon(\tau_1), \ldots, \xi_1 : \texttt{of}\ x_n\ \varepsilon(\tau_n)\}$.

Theorem B.7 Adequacy for type assignment deductions: ε *is a bijection between the typing deductions* $\mathcal{T} :: X \vdash e \in \tau$ *and the canonical forms of type* of $\varepsilon_{p(X)}(e)\ \varepsilon(\tau)$ *in the signature* Σ_{TP} *and the context* Γ_X. *The encoding is compositional, i.e., for a typing deduction* $\mathcal{T} :: X \vdash e \in \tau$ *where* $X = \{\xi_1 :: x_1 \in \tau_1, \ldots, \xi_n :: x_n \in \tau_n\}$ *and typing deductions* $\mathcal{T}_1 :: X' \vdash e_1 \in \tau_1, \ldots, \mathcal{T}_n :: X' \vdash e_n \in \tau_n$

$$\varepsilon([\mathcal{T}_1/\xi_1, \ldots, \mathcal{T}_n/\xi_n]\mathcal{T}) = [\varepsilon(\mathcal{T}_1)/\xi_1, \ldots, \varepsilon(\mathcal{T}_n)/\xi_n]\varepsilon(\mathcal{T})$$

Although there are more details to be handled, adequacy for extraction and extraction/simplification deductions can be stated similarly.

Appendix C

An Elf program to recognize tail-recursive form

The following program encodes a deductive system for establishing whether a program is in tail-recursive form. The key idea of the deduction is to check whether a given bound variable occurs free in certain subterms of a recursive definition. There are three judgments of the system. The judgment `tl_rec_pgm` holds if a program expression is tail-recursive as a whole. The judgment `tl_rec_def` holds if a primitive recursive function definition is tail-recursive. Finally, `tl_rec_body` is defined on functions from program terms to program terms, and does the real work of checking that the bound variable represented by the function parameter occurs only in legal positions. Note that there are two rules for the case where the construct `app (lam M) N` occurs. This corresponds to a let-binding, where the function parameter of interest must be restricted to occur in the binder or in the body, but not both.

```
tl_rec_pgm : term -> type.
tl_rec_def : term -> type.
tl_rec_body : (term -> term) -> type.

trbody_id : tl_rec_body ([x] x).
trbody_app : tl_rec_body ([x] app (M x) N)
     <- tl_rec_body M.
trbody_app_lam1 : tl_rec_body ([x] app (lam M) (N x))
     <- tl_rec_body ([x] M (N x)).
trbody_app_lam2 : tl_rec_body ([x] app (lam (M x)) N)
     <- tl_rec_body ([x] M x N).
trbody_lam : tl_rec_body ([x] lam [y] M x y)
     <- ({y} tl_rec_body ([x] M x y)).
trbody_case : tl_rec_body ([x] decide M (Nl x) (Nr x))
     <- ({y} tl_rec_body ([x] Nl x y))
     <- ({y} tl_rec_body ([x] Nr x y)).
trbody_if : tl_rec_body ([x] if M (N0 x) (N1 x))
     <- tl_rec_body ([x] N0 x)
     <- tl_rec_body ([x] N1 x).

trdef_nat_ind : tl_rec_def (nat_ind Mb Ms)
```

```
     <- tl_rec_pgm Mb
     <- ({x} tl_rec_body ([y] Ms x y)).
trdef_list_ind : tl_rec_def (list_ind Mnil Mcons)
     <- tl_rec_pgm Mnil
     <- ({l} {x} tl_rec_body ([y] Mcons l y x)).

trpgm_nat_ind : tl_rec_pgm (nat_ind Mb Ms)
     <- tl_rec_def (nat_ind Mb Ms).
trpgm_list_ind : tl_rec_pgm (list_ind Mnil Mcons)
     <- tl_rec_def (list_ind Mnil Mcons).
trpgm_app : tl_rec_pgm (app M N)
     <- tl_rec_pgm M
     <- tl_rec_pgm N.
trpgm_lam : tl_rec_pgm (lam M)
     <- ({x} tl_rec_pgm x -> tl_rec_pgm (M x)).
trpgm_unity : tl_rec_pgm unity.
trpgm_0 : tl_rec_pgm 0.
trpgm_s : tl_rec_pgm (s M)
     <- tl_rec_pgm M.
trpgm_pr : tl_rec_pgm (pair M N)
     <- tl_rec_pgm M
     <- tl_rec_pgm N.
trpgm_fst : tl_rec_pgm (fst M)
     <- tl_rec_pgm M.
trpgm_snd : tl_rec_pgm (snd M)
     <- tl_rec_pgm M.
trpgm_spread : tl_rec_pgm (spread M N)
     <- tl_rec_pgm M
     <- ({x} {y} tl_rec_pgm x -> tl_rec_pgm y
         -> tl_rec_pgm (N x y)).
trpgm_inl : tl_rec_pgm (inl M)
     <- tl_rec_pgm M.
trpgm_inr : tl_rec_pgm (inr M)
     <- tl_rec_pgm M.
trpgm_decide : tl_rec_pgm (decide M Nl Nr)
     <- tl_rec_pgm M
     <- ({x} tl_rec_pgm x -> tl_rec_pgm (Nl x))
     <- ({x} tl_rec_pgm x -> tl_rec_pgm (Nr x)).
trpgm_any : tl_rec_pgm (any M)
     <- tl_rec_pgm M.
trpgm_neg : tl_rec_pgm neg.
trpgm_case_nat : tl_rec_pgm (case_nat M N0 Ns)
     <- tl_rec_pgm M
     <- tl_rec_pgm N0
     <- {x} tl_rec_pgm x -> tl_rec_pgm (Ns x).
trpgm_pred : tl_rec_pgm (pred M)
     <- tl_rec_pgm M.
trpgm_minus : tl_rec_pgm (M - N)
     <- tl_rec_pgm M
     <- tl [illegible].
trpgm_plus   tl_rec_pgm (M + N)
```

```
     <- tl_rec_pgm M
     <- tl_rec_pgm N.
trpgm_mul : tl_rec_pgm (mul M N)
     <- tl_rec_pgm M
     <- tl_rec_pgm N.
trpgm_sign : tl_rec_pgm (signum M)
     <- tl_rec_pgm M.
trpgm_eq : tl_rec_pgm (M = N)
     <- tl_rec_pgm M
     <- tl_rec_pgm N.
trpgm_if : tl_rec_pgm (if M Nt Nf)
     <- tl_rec_pgm M
     <- tl_rec_pgm Nt
     <- tl_rec_pgm Nf.
trpgm_ge : tl_rec_pgm (M >= N)
     <- tl_rec_pgm M
     <- tl_rec_pgm N.
trpgm_eq? : tl_rec_pgm (=? M N)
     <- tl_rec_pgm M
     <- tl_rec_pgm N.
trpgm_nil : tl_rec_pgm nil.
trpgm_cons : tl_rec_pgm (cons M N)
     <- tl_rec_pgm M
     <- tl_rec_pgm N.
trpgm_case_list : tl_rec_pgm (case_list M Nnil Ncons)
     <- tl_rec_pgm M
     <- tl_rec_pgm Nnil
     <- {x} tl_rec_pgm x -> {l} tl_rec_pgm l
        -> tl_rec_pgm (Ncons x l).
trpgm_@ : tl_rec_pgm (@ M N)
     <- tl_rec_pgm M
     <- tl_rec_pgm N.
```

Using the techniques of Chapter 3, we partially encoded a proof that the tail-recursion introduction transformation of Chapter 4 yields a tail-recursive program if it succeeds.

Theorem C.1 *Any program extracted from the output proof constructed by a successful application of tail-recursion introduction is tail-recursive.*

The theorem is represented by the declaration:

```
tl_rec_pf :
 tail_rec_doubleneg _ _ _ _ _ Phi Psi
  Input_proof I1 I2 I3
  Output_proof ->
 extract_simp Output_proof Prog -> tl_rec_pgm Prog -> type.
```

The proof relies on "inversion": since the transformation is defined by a single clause, the form of the output proof can be read off from that clause. This in turn determines the form of the extraction deduction, thus of the extracted program, in enough detail that the required deduction of tail-recursiveness can be constructed. The representation of the proof is incomplete because we have assumed without proof that any program expression extracted from an individual term of the logic is tail-recursive. This is not hard to prove although the details are tedious. The following clause represents the partial proof:

```
tr_pf :
 {R0} {S0} {MS0} {ES0 : {k} {k'} extract_l_tm k k' -> extract_l_tm (S0 k) (MS0 k')}
 {K0} {MK0} {EK0 : extract_l_tm K0 MK0}
 {F}
 {H} {Mh}
 {EH : {x} {x'} extract_tm x x' -> {k} {k'} extract_l_tm k k' -> extract_l_tm (H x k) (Mh x' k')}
 {Phi} {Psi} {Db} {I1} {I2} {I3}
 {Ds} {Nnbase} {Nnstep} {Nconc} {Nnc_ded} {Nns_ded} {Nnb_ded}
 {base_trpf : {k} tl_rec_pgm k -> tl_rec_pgm (MS0 k)}
 {init_trpf : tl_rec_pgm MK0}
 tl_rec_pf
  (tl_rec_doubleneg R0 S0 K0 F H Phi Psi Db I1 I2 I3
   Ds Nnbase Nnstep Nconc
   Nnc_ded Nns_ded Nnb_ded)
  % Extraction deduction must have the following form:
  (exs_lforalli
   ([l1] [l1'] [el1: extract_l_tm l1 l1']
    exs_lexistse1
     (exs_lforalle EK0
      (exs_lforalle el1
       (exs_lind ([l] [l'] [el: extract_l_tm l l']
        [p] [p'] [ep: extract_simp p p']
        [x] [x'] [ex: extract_tm x x']
        exs_lforalli ([k] [k'] [ek: extract_l_tm k k']
         exs_lexistse1 (exs_lforalle (EH x x' ex k k' ek) ep (inf_lforall [k] inf_lexists))
          ([s] [s'] [es: extract_l_tm s s']
           [hyp] [ehyp: extract_simp hyp unity]
           exs_lexistsi1 es ([x] unin_n))
           ([l] unin_n))
         (inf_lforall [l] inf_lexists))
        (exs_lforalli
         ([k] [k'] [ek: extract_l_tm k k'] exs_lexistsi1 (ES0 k k' ek) ([x] unin_n))
         (inf_lforall [l] inf_lexists))
        ([l] inf_lforall [l] inf_lexists))
       (inf_lforall [l] inf_lforall [k] inf_lexists))
      (inf_lforall [k] inf_lexists))
     ([s] [s'] [es: extract_l_tm s s'] [p] [ep: extract_simp p unity]
      exs_lexistsi1 es ([s] unin_n))
     ([l:ilist] unin_n))
   (inf_lforall ([l] inf_lexists)))
  % Deduction that extracted program is tail-recursive:
  (trpgm_lam
  ([l] [trl]
trpgm_app
 (trpgm_app
  init_trpf
  (trpgm_app
   trl
   (trpgm_list_ind
    (trdef_list_ind
     ([l] [x] trbody_lam ([y] trbody_app_lam1 (trbody_app trbody_id)))
     (trpgm_lam [k] [trk] base_trpf k trk) ))))
 (trpgm_lam ([x] [trx] trx)))) .
```

Appendix D

Extensions of Elf encoding to many-sorted logic for Chapter 4

D.1 Extended arithmetic

```
%% Interpret "i" of FOL as natural numbers

zero : i.
succ : i -> i.

%% Equality
eq :  i -> i -> o.

ax_zero : {X:i} |- (not (eq (succ X) zero)).

eq_refl : {T:i} |- (eq T T).
eq_repl : {A:i -> o} |- (A T) -> |- (eq T T') -> |- (A T').

% Induction

ind : {A:i -> o} |- (A zero) -> ({x:i} |- (A x) -> |- (A (succ x)))
        -> |- (forall A).

% predecessor, subtraction, addition, multiplication

prd : i -> i.

prdz : |- (eq (prd zero) zero).
prds : {T} |- (eq (prd (succ T)) T).

sb : i -> i -> i.

sbz : {T} |- (eq (sb T zero) T).
sbs : {T1} {T2} |- (eq (sb T1 (succ T2)) (prd (sb T1 T2))).
```

```
ad : i -> i -> i.

adz : {T} |- (eq (ad T zero) T).
ads : {T1} {T2} |- (eq (ad T1 (succ T2)) (succ (ad T1 T2))).

ml : i -> i -> i.

mlz : {T} |- (eq (ml T zero) zero).
mls : {T1} {T2}|- (eq (ml T1 (succ T2)) (ad (ml T1 T2) T1)).

%% "Boolean" functions (zero codes truth, (succ zero) codes falsehood)

sg : i -> i.            % (sg x) = 0 iff x = 0, (sg x) <= 1 for all x

sgx : {T} |- (eq (sg T) (sb (succ zero) (sb (succ zero) T))).

% equality test

eq? : i -> i -> i.

eq?x : {T1} {T2} |- (eq (eq? T1 T2) (sg (ad (sb T1 T2) (sb T2 T1)))).

% conditionals

switch : i -> i -> i -> i.

switchz : {T1} {T2} |- (eq (switch zero T1 T2) T1).
switchs : {T1} {T2} |- (eq (switch (succ zero) T1 T2) T2).

% inequalities

ge? : i -> i -> i.              % greater than or equal

ge?x : {T1} {T2} |- (eq (ge? T1 T2) (sg (sb T2 T1))).

%%% Extraction with removal of uninformative parts.
%%% For primitive recursive arithmetic rules of proof.

%%% DYNAMIC

%% Equality

% Peq proves (eq T1 T2) thus has no computational content.
exs_eq_repl : extract_simp (eq_repl A Pa Peq) M
               <- extract_simp Pa M.

%% UNIVERSAL QUANTIFICATION
exs_foralli : extract_simp ((foralli P):|- (forall A)) (lam M)
              <- inf (forall A)
              <- {X:i} {x:term} (extract_tm X x -> extract_simp (P X) (M x)).
```

```
exs_foralle : extract_simp (foralle T (P: |- (forall A))) (app M N)
             <- inf (forall A)
             <- extract_simp P M <- extract_tm T N.

%% EXISTENTIAL QUANTIFICATION
%% Don't build (pair M N:A) if A is uninformative.
exs_existsi1 : extract_simp (existsi A T _) M
               <- ({x:i} uninf (A x)) <- extract_tm T M.
exs_existsi2 : extract_simp (existsi A T P) (pair M N)
               <- ({x:i} inf (A x))
               <- extract_tm T M <- extract_simp P N.

%% We don't have (pair M N:A) if A is uninformative.
exs_existse1 : extract_simp (existse P (Q: |- (exists A))) (app (lam M) N)
               <- ({X:i} uninf (A X))
               <- ({X:i} {x:term} extract_tm X x
                   -> {p: |- (A X)} extract_simp (P X p) (M x))
               <- extract_simp Q N.

exs_existse2 : extract_simp (existse P_min P_maj) (spread N M)
               <- ({X:i} inf (A X))
               <- ({X:i} {x:term} extract_tm X x
                   -> {P:|- (A X)} {p:term} extract_simp P p
                   -> extract_simp (P_min X P) (M x p))
               <- extract_simp P_maj N.

%% INDUCTION
exs_ind : extract_simp (ind A Pz Ps) (nat_ind Nz Ns)
          <- ({X} inf (A X))
          <- extract_simp Pz Nz
          <- {X:i} {x:term} extract_tm X x
              -> {P:|- (A X)} {p:term} extract_simp P p
              -> extract_simp (Ps X P) (Ns x p).
```

D.2 Monomorphic lists

```
%%% (Logical) syntactic categories.

ilist : type.                    %name ilist l k u v w l' k' u' v' w'

%%% (Logical) abstract syntax.

lforall : (ilist -> o) -> o.
lexists : (ilist -> o) -> o.

%%% Lists for FOL

nl : ilist.
cns : i -> ilist -> ilist.
```

```
leq : ilist -> ilist -> o.

leq_refl : {L:ilist} |- (leq L L).
leq_repl : {A:ilist -> o} |- (A L) -> |- (leq L L') -> |- (A L').

leq_nl : {X:i} {L:ilist} |- (not (leq nl (cns X L))).
leq_cns : |- (eq X Y) -> |- (leq L1 L2) -> |- (leq (cns X L1) (cns Y L2)).

%%% Rules of proof for quantifiers.

lforalli : ({x:ilist} |- (A x)) -> |- (lforall A).
lforalle : {T:ilist} |- (lforall A) -> |- (A T).

lexistsi : {A:ilist -> o} {T:ilist} |- (A T) -> |- (lexists A).
lexistse : ({x:ilist} |- (A x) -> |- C) -> |- (lexists A) -> |- C.

lind : {A:ilist -> o} |- (A nl)
          -> ({l:ilist} |- (A l) -> {x:i} |- (A (cns x l)))
           -> |- (lforall A).

elem : i -> ilist -> o.
elem_nil : {X} |- (not (elem X nl)).
elem_hd : {X} {L} |- (elem X (cns X L)).
elem_tail : {Y} |- (elem X L) -> |- (elem X (cns Y L)).
elem_ind : {A:i -> o} {Y} {X} {L} |- (elem Y (cns X L))
            -> |- (A X)
            -> ({Z} |- (elem Z L) -> |- (A Z))
            -> |- (A Y).
elem_case : |- (not (eq X Y)) -> |- (not (elem X L)) -> |- (not (elem X (cns Y L))).

elem_decide : {X} {L} |- (or (elem X L) (not (elem X L))).

append : ilist -> ilist -> ilist.
append_id_l : {L} |- (leq (append nl L) L).
append_id_r : {L} |- (leq (append L nl) L).
append_assoc : {X} {L1} {L2} |- (leq (append (cns X L1) L2) (cns X (append L1 L2))).

lswitch : i -> ilist -> ilist -> ilist.

lswitchz : {T1} {T2} |- (leq (lswitch zero T1 T2) T1).
lswitchs : {T1} {T2} |- (leq (lswitch (succ zero) T1 T2) T2).

%%%
%%% Extraction/simplification
%%%

h_leq : harrop (leq L K).
h_elem : harrop (elem X L).

h_lforall : harrop (lforall A) <- {x:ilist} harrop (A x).
```

```
unin_leq : uninf (leq L K).
unin_elem : uninf (elem X L).

unin_lforall : uninf (lforall A) <- {x:ilist} uninf (A x).

inf_lexists : inf (lexists A).

inf_lforall : inf (lforall A) <- {x:ilist} inf (A x).


extract_l_tm : ilist -> term -> type.

ex_nil : extract_l_tm nl nil.
ex_cons : extract_l_tm (cns X L) (cons X' L')
          <- extract_tm X X' <- extract_l_tm L L'.
ex_append : extract_l_tm (append L1 L2) (@ L1' L2')
           <- extract_l_tm L1 L1' <- extract_l_tm L2 L2'.
ex_lif : extract_l_tm (lswitch X L1 L2) (if M M1 M2)
        <- extract_tm X M
        <- extract_l_tm L1 M1
        <- extract_l_tm L2 M2.

%%% Extraction, with simplification, for lists of individuals

%%% DYNAMIC

%% Equality

% Peq proves (leq T1 T2) thus has no computational content.
exs_leq_repl : extract_simp (leq_repl A Pa Peq) M
               <- extract_simp Pa M.

%% Membership

% P_elem has no computational content
exs_elem_ind : extract_simp (elem_ind A Y X L P_elem P_hd P_tl)
                (if (Y' = X') N1 (Nr Y'))
               <- extract_tm Y Y' <- extract_tm X X'
               <- extract_simp P_hd N1
               <- ({T} {t} extract_tm T t
                  -> {P} extract_simp (P_tl T P) (Nr t)).

%% UNIVERSAL QUANTIFICATION
exs_lforalli : extract_simp ((lforalli P):|- (lforall A)) (lam M)
               <- inf (lforall A)
               <- {L:ilist} {l:term} (extract_l_tm L l -> extract_simp (P L) (M l)).

exs_lforalle : extract_simp (lforalle T (P: |- (lforall A))) (app M N)
               <- inf (lforall A)
               <- extract_simp P M <- extract_l_tm T N.
```

```
%% EXISTENTIAL QUANTIFICATION
%% Don't build (pair M N:A) if A is uninformative.
exs_lexistsi1 : extract_simp (lexistsi A T _) M
                <- ({x:ilist} uninf (A x)) <- extract_l_tm T M.

exs_lexistsi2 : extract_simp (lexistsi A T P) (pair M N)
                <- ({x:ilist} inf (A x))
                <- extract_l_tm T M <- extract_simp P N.

%% We don't have (pair M N:A) if A is uninformative.
exs_lexistse1 : extract_simp (lexistse P (Q: |- (lexists A))) (app (lam M) N)
                <- ({L:ilist} uninf (A L))
                <- ({L:ilist} {l:term} extract_l_tm L l
                    -> {p: |- (A L)} extract_simp p unity
                    -> extract_simp (P L p) (M l))
                <- extract_simp Q N.

exs_lexistse2 : extract_simp (lexistse P_min P_maj) (spread N M)
                <- ({L:ilist} inf (A L))
                <- ({L:ilist} {l:term} extract_l_tm L l
                    -> {P:|- (A L)} {p:term} extract_simp P p
                    -> extract_simp (P_min L P) (M l p))
                <- extract_simp P_maj N.

exs_lind : extract_simp (lind A Pn Pl) (list_ind Nn Nl)
           <- ({L} inf (A L))
           <- extract_simp Pn Nn
           <- ({L:ilist} {l:term} extract_l_tm L l
               -> {P:|- (A L)} {p:term} extract_simp P p
               -> {X:i} {x:term} extract_tm X x
               -> extract_simp (Pl L P X) (Nl l p x)).
```

D.3 Functional programming language

```
%%% Abstract syntax of terms and types.

%%% Static.

% Extensions for nats.

pred : term -> term.
- : term -> term -> term.       %infix left 20 -

+ : term -> term -> term.       %infix left 20 +

mul : term -> term -> term.
```

```
signum : term -> term.

= : term -> term -> term.        %infix none 10 =

if : term -> term -> term -> term.

>= : term -> term -> term.       %infix none 10 >=
=? : term -> term -> term.

% Extensions for lists of nats.

list : tp.
nil : term.
cons : term -> term -> term.
list_ind : term -> (term -> term -> term -> term) -> term.
case_list : term -> term -> (term -> term -> term ) -> term.
@ : term -> term -> term.
```

Appendix E

Elf encodings of extracted programs of Chapter 4

This appendix gives the programs of Chapter 4 as extracted by the implementation. These differ from the ML versions not only in syntax but in the absence of let-bindings.

The code extracted from Proof 4.2:

```
list_ind nil
         ([l:term] [p:term] [x:term]
          app (lam [l1:term] if (x >= s s 0) (cons x l1) l1) p)
```

The code extracted from the result of applying the naive form of tail-recursion introduction:

```
 lam [l:term]
  spread
   (app (app (list_ind
              (lam [l1:term]
                pair l1
                 (pair nil
                   (app (lam [l11:term] lam [x:term] inr unity) l1)))
              ([l1:term] [p:term] [x:term] lam [l11:term]
                spread (app p (if (x >= s s 0) (cons x l11) l11))
                 ([l111:term] [p1:term]
                   spread p1
                    ([l1111:term] [p11:term]
                      pair l111
                       (pair (if (x >= s s 0) (cons x l1111) l1111)
                         (app (app (app
 (app (app (lam [x1:term] lam [l11111:term] lam [l111111:term]
           lam [l1111111:term] lam [q:term] lam [x11:term]
           decide (app q x11) ([p111:term] inl unity)
```

```
             ([p111:term]
               decide (if (x1 >= s s 0) (inl unity) (inr unity))
                ([p1111:term] if (x11 = x1) (inl unity) (inr unity))
                ([p1111:term] inr unity)))
            x)
        l11)
   l1111)
                                            l111)
                                       p11)))))
             l)
        nil)
   ([l1:term] [p:term] spread p ([l11:term] [p1:term] l1))
```

The code extracted from the result of applying the lifted form of tail-recursion introduction:

```
lam [l:term]
 spread
  (app (app (list_ind
              (lam [l1:term]
                pair l1 (pair nil (lam [x:term] inr unity)))
              ([l1:term] [p:term] [x:term] lam [l11:term]
                spread (app p (if (x >= s s 0) (cons x l11) l11))
                 ([l111:term] [p1:term]
                   spread p1
                    ([l1111:term] [p11:term]
                      pair l111
                       (pair (if (x >= s s 0) (cons x l1111) l1111)
                         (lam [x1:term]
                           decide (app p11 x1)
                            ([p111:term] inl unity)
                            ([p111:term]
                              decide
                               (if (x >= s s 0) (inl unity)
                                   (inr unity))
                               ([p1111:term]
                                 if (x1 = x) (inl unity) (inr unity))
                               ([p1111:term] inr unity))))))))
             l)
        nil)
   ([l1:term] [p:term] spread p ([l11:term] [p1:term] l1))
```

The code extracted from the result of applying lifted tail-recursion introduction with double negation:

```
lam [l:term]
 app (lam [l1:term] l1)
     (app (app (list_ind (lam [l1:term] l1)
                    ([l1:term] [p:term] [x:term] lam [l11:term]
                      app (lam [l111:term] l111)
                          (app p (if (x >= s s 0) (cons x l11) l11)))) l)
           nil)
```

References

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman.
Data Structures and Algorithms.
Addison-Wesley, 1983.

[Bar91] Henk Barendregt.
Introduction to generalized type systems.
Journal of Functional Programming, 1(2):125–154, April 1991.

[BC85] Joseph Bates and Robert Constable.
Proofs as programs.
ACM Transactions on Programming Languages and Systems, 7(1):113–136, January 1985.

[BC93] David A. Basin and Robert L. Constable.
Metalogical frameworks.
In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 1–29. Cambridge University Press, 1993.
Also available as Max-Planck-Institut für Informatik technical report MPI-I-92-205.

[BD77] R. M. Burstall and John Darlington.
A transformation system for developing recursive programs.
Journal of the Association for Computing Machinery, 24(1):44–67, January 1977.

[Bir84] R.S. Bird.
The promotion and accumulation strategies in transformational programming.
ACM Transactions on Programming Languages and Systems, 6(4):487–504, October 1984.

[BM79] Robert S. Boyer and J. Strother Moore.
A Computational Logic.
ACM monograph series. Academic Press, New York, 1979.

[C$^+$86] Robert L. Constable et al.
Implementing Mathematics with the Nuprl Proof Development System.
Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[CH85] Thierry Coquand and Gérard Huet.
Constructions: A higher order proof system for mechanizing mathematics.
In *EUROCAL85.* Springer-Verlag LNCS 203, 1985.

[dB80] N. G. de Bruijn.
A survey of the project Automath.
In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[Des86] Joëlle Despeyroux.
Proof of translation in natural semantics.
In A. R. Meyer, editor, *Symposium on Logic in Computer Science*, pages 193–205, Cambridge, Massachusetts, June 1986. IEEE Computer Society Press.

[DF92] Olivier Danvy and Andrzej Filinski.

Representing control: A study of the CPS transformation.
Mathematical Structures in Computer Science, 2(4):361–391, December 1992.

[DFH+91] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Christine Paulin-Mohring, and Benjamin Werner.
The Coq proof assistant user's guide.
Rapport Techniques 134, INRIA, Rocquencourt, France, December 1991.
Version 5.6.

[Fel89] Amy Felty.
Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language.
PhD thesis, Department of Computer and Information Science, University of Pennsylvania, July 1989.
Available as Technical Report MS-CIS-89-53.

[Gen69] Gerhard Gentzen.
Investigations into logical deductions.
In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland Publishing Co., Amsterdam, 1969.

[Goa80] Christopher A. Goad.
Computational uses of the manipulation of formal proofs.
Technical Report Stan-CS-80-819, Stanford University, August 1980.

[Han91] John Hannan.
Investigating a Proof-Theoretic Meta-Language for Functional Programs.
PhD thesis, University of Pennsylvania, January 1991.
Available as MS-CIS-91-09.

[Har60] Ronald Harrop.
Concerning formulas of the types $A \rightarrow B \vee C, A \rightarrow (Ex)B(x)$ in intuitionistic formal systems.
Journal of Symbolic Logic, 25(1):27–32, March 1960.

[Hay90] Susumu Hayashi.
An introduction to PX.
In Gerard Huet, editor, *Logical Foundations of Functional Programming.* Addison-Wesley, 1990.

[Hey34] A. Heyting.
Mathematische Grundlagenforschung. Intuitionismus. Beweistheorie.
Springer, 1934.
Cited in [TvD88]. Reprinted 1974.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin.
A framework for defining logics.
Journal of the Association for Computing Machinery, 40(1):143–184, January 1993.

[HL78] Gérard Huet and Bernard Lang.
Proving and applying program transformations expressed with second-order patterns.
Acta Informatica, 11:31–55, 1978.

[HM88] John Hannan and Dale Miller.
Uses of higher-order unification for implementing program transformers.
In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 2*, pages 942–959, Cambridge, Massachusetts, August 1988. MIT Press.

[HM89] John Hannan and Dale Miller.
A meta-logic for functional programming.
In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming*, pages 453–476. MIT Press, 1989.

[How69] W. A. Howard.
The formulae-as-types notion of construction.
Unpublished manuscript, 1969.
Published in To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, 1980.

[HP99] Robert Harper and Frank Pfenning.
A module system for a programming language based on the LF logical framework.
Journal of Functional Programming, 199?
To appear.

[HP92] John Hannan and Frank Pfenning.
Compiler verification in LF.
In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[Kah87] Gilles Kahn.
Natural semantics.
In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag LNCS 247, 1987.

[LPT89] Zhaohui Luo, Robert Pollack, and Paul Taylor.
How to use LEGO.
Technical Report LFCS–TN–27, Laboratory for Foundations of Computer Science, University of Edinburgh, 1989.

[Mad91] Peter Madden.
Automated Program Transformation Through Proof Transformation.
PhD thesis, University of Edinburgh, 1991.

[Mag92] Lena Magnusson.
The new implementation of ALF.
In B. Nordström, K. Petersson, and G. Plotkin, editors, *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, pages 265–282, Båstad, Sweden, June 1992. University of Göteborg.

[Mil91] Dale Miller.
A logic programming language with lambda-abstraction, function variables, and simple unification.

In Peter Schroeder-Heister, editor, *Extensions of Logic Programming: International Workshop, Tübingen FRG, December 1989*, pages 253–281. Springer-Verlag LNCS 475, 1991.

[ML73] Per Martin-Löf.
An intuitionistic theory of types: Predicative part.
In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118. North-Holland, 1973.

[ML80] Per Martin-Löf.
Constructive mathematics and computer programming.
In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holland, 1980.

[MP91] Spiro Michaylov and Frank Pfenning.
Natural semantics and some of its meta-theory in Elf.
In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596.

[Mur90] Chetan Murthy.
Extracting Constructive Content from Classical Proofs.
PhD thesis, Cornell University, August 1990.

[MW81] Zohar Manna and Richard Waldinger.
Deductive synthesis of the unification algorithm.
Science of Computer Programming, 1:5–48, 1981.

[Pfe89] Frank Pfenning.
Elf: A language for logic definition and verified meta-programming.
In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[Pfe90] Frank Pfenning.
Program development through proof transformation.
Contemporary Mathematics, 106:251–262, 1990.

[Pfe91a] Frank Pfenning.
Logic programming in the LF logical framework.
In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe91b] Frank Pfenning.
Unification and anti-unification in the Calculus of Constructions.
In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[Pfe92a] Frank Pfenning.
Computation and deduction.
Unpublished Lecture Notes, May 1992.

[Pfe92b] Frank Pfenning.
Intersection types for a logical framework.

POP Report 92-006, School of Computer Science, Carnegie Mellon University, December 1992.

[PK80] Robert Paige and Shaye Koening.
Finite differencing of computable expressions.
Technical Report LCSR-TR-8, Laboratory for Computer Science Research, Rutgers University, August 1980.

[Plo75] G. D. Plotkin.
Call-by-name, call-by-value and the λ-calculus.
Theoretical Computer Science, 1:125–159, 1975.

[PM89] Christine Paulin-Mohring.
Extracting F_ω programs from proofs in the calculus of constructions.
In *Sixteenth Annual Symposium on Principles of Programming Languages*, pages 89–104. ACM Press, January 1989.

[PM93] Christine Paulin-Mohring.
Inductive definitions in the system Coq: Rules and properties.
In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*, pages 328–345, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664.

[PR92] Frank Pfenning and Ekkehard Rohwedder.
Implementing the meta-theory of deductive systems.
In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 537–551, Saratoga Springs, New York, June 1992. Springer-Verlag LNAI 607.

[Pra71] Dag Prawitz.
Ideas and results in proof theory.
In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 235–307, Amsterdam, London, 1971. North-Holland Publishing Co.

[PW90] Christine Paulin and Benjamin Werner.
Extracting and executing programs developed in the inductive constructions system: a progress report.
In G. Huet and G. Plotkin, editors, *Proceedings of the First Workshop on Logical Frameworks*, pages 377–390. Preliminary Version, May 1990.

[Raf93] Christophe Raffalli.
Machine deduction.
To appear in the Proceedings of the Workshop on Types for Proofs and Programs, Nijmegen, The Netherlands, May 1993.

[Sas86] James T. Sasaki.
Extracting Efficient Code from Constructive Proofs.
PhD thesis, Cornell University, May 1986.
Available as Technical Report TR 86-757.

[Sch82] Helmut Schwichtenberg.
On Martin-Löf's theory of types.

In *Atti Degli Incontri di Logica Mathematica*, pages 299–325. Dipartimento di Matematica, Università di Siena, 1982.

[Sch85] Helmut Schwichtenberg.
A normal form for natural deductions in a type theory with realizing terms.
In Ettore Casari et al., editors, *Atti del Congresso Logica e Filosfia della Scienza, oggi. San Gimignano, December 7–11, 1983*, pages 95–138, Bologna, Italy, 1985. CLUEB.

[Smi91] Douglas R. Smith.
KIDS – a knowledge-based software development system.
In Michael R. Lowry and Robert D. McCartney, editors, *Automating Software Design*, pages 483–514. AAAI Press/MIT Press, 1991.

[SS83] William L. Scherlis and Dana S. Scott.
First steps towards inferential programming.
In R.E.A. Mason, editor, *Information Processing*, pages 199–212. Elsevier Science Publishers, 1983.

[SW93] Wilfred Sieg and Stanley Wainer.
Personal communication from S. Wainer, September 1993.

[TvD88] A. S. Troelstra and D. van Dalen.
Constructivism in Mathematics, volume 121 of *Studies in Logic and the Foundations of Mathematics*.
North-Holland, Amsterdam, 1988.